Defence Research and Development Canada
Recherche et développement pour la défense Canada

DEFENCE R&D DÉFENSE

Opening up architectures of software-intensive systems

A first prototype implementation

P. Charland
D. Dessureault
D. Ouellet
M. Lizotte
DRDC Valcartier

Defence R&D Canada – Valcartier
Technical Memorandum
DRDC Valcartier TM 2006-781
November 2007

Canada

Opening up architectures of software-intensive systems

A first prototype implementation

P. Charland
D. Dessureault
D. Ouellet
M. Lizotte
DRDC Valcartier

Defence R&D Canada - Valcartier

Technical Memorandum

DRDC Valcartier TM 2006-781

November 2007

Author

Philippe Charland

Approved by

Guy Turcotte
Head System of Systems

Approved for release by

Christian Carrier
Chief Scientist

Abstract

Although there already exist tools to assist in understanding the behavior of software systems when no complete and consistent design models are available, these tools generate a large volume of information. One approach to deal with this problem is information hiding. This technical memorandum presents a prototype which implements this technique to reverse engineer dynamic models from Java software systems. These models are represented using Unified Modeling Language (UML) sequence diagrams. Such diagrams show the interactions, in terms of messages or information transfers, between the operational nodes of a system, arranged in a time sequence. Information hiding is achieved by reconstructing the sequence diagrams at various levels of abstraction. The interactions between the operational nodes of a system can be displayed at a low level, i.e., object level. However, related operational nodes can also be regrouped into higher level structures, i.e., packages. The proposed approach was implemented in Eclipse, an extensible integrated development environment (IDE). The objective is to complement the behavioral views reverse engineered by the implemented prototype with structure views generated by other tools.

Résumé

Bien qu’il existe déjà des outils pour aider à comprendre le fonctionnement de systèmes logiciels lorsqu’aucun modèle de conception complet et cohérent n’est disponible, il reste que ces outils génèrent une grande quantité d’information. Une des approches pour résoudre ce problème est le masquage d’information. Ce mémorandum technique présente un prototype qui implante cette technique pour faire la rétro-ingénierie de modèles dynamiques à partir de systèmes logiciels développés en Java. Ces modèles sont représentés sous forme de diagrammes de séquence UML (Unified Modeling Language). De tels diagrammes montrent les interactions, en termes de messages ou de transferts d'information, disposées de façon séquentielle, entre les nœuds opérationnels d'un système. Le masquage d’information est accompli en reconstruisant les diagrammes à différents niveaux d’abstraction. Les interactions entre les nœuds opérationnels d’un système peuvent être affichées à un bas niveau, c.-à-d. au niveau des objets. Cependant, les nœuds opérationnels rattachés peuvent aussi être regroupés dans des structures de plus haut niveau, c.-à-d. des paquetages. L’approche proposée a été implantée dans Eclipse, un environnement de développement intégré (EDI) extensible. L’objectif est de compléter les vues de fonctionnement obtenues du prototype par rétro-ingénierie par des vues de structures générées par d’autres outils.

This page intentionally left blank.

Executive Summary

To understand an existing object-oriented software system, information relating to its structure and behavior is required. When there is no complete nor consistent design models available, one has to fall back on reverse engineering to retrieve as much information as possible through static and dynamic analysis. Capacities to reverse engineer the static structure of object-oriented systems are already present in several Computer-Aided Software Engineering (CASE) tools, although some issues still need to be addressed. Reverse engineering the behavior of an object-oriented system is even more difficult than understanding its structure, due to the specific characteristics of the object-oriented paradigm, such as inheritance, polymorphism and dynamic binding. These peculiarities make it difficult to comprehend the system behavior using code analysis only. As a result, dynamic models must be extracted by means of dynamic analysis. Even though several approaches have been proposed for the extraction of dynamic models for object-oriented systems, one problem faced is the large volume of information generated in the execution traces.

This technical memorandum presents a prototype developed by the members of the Opening up Architectures of Software-Intensive Systems (OASIS) project to address the trace explosion problem through information hiding. This prototype, implemented as plug-ins in the Eclipse Integrated Development Environment (IDE), allows reverse engineering dynamic models from Java software systems. These models are represented using Unified Modeling Language (UML) sequence diagrams. Such diagrams show the interactions, in terms of messages or information transfers, between the operational nodes of a system, arranged in a time sequence. Information hiding is achieved by reconstructing the sequence diagrams at various levels of abstraction. The interactions between the operational nodes of a system can be displayed at a low level, i.e., object level. However, related operational nodes can also be regrouped into higher level structures, i.e., packages. This level allows a considerable reduction of the diagrams size, as the messages exchanged between objects of the same package are encapsulated in the latter and not displayed in the diagram. Also, the package level assists in mapping system functionalities to architectural elements, something which is essential when trying to understand an unfamiliar system at the architectural level.

Following the implementation of this prototype, another one, with an improved set of functionalities as well as additional views, should be developed to assist the Canadian Forces (CF) in recovering and comprehending the architectures of their legacy software systems. Ideally, once this prototype is developed, a qualitative study should be performed. Its objective would be to assess the added value of the OASIS architecture recovery and comprehension prototype on the understanding of participants.

Charland, P., Dessureault, D., Ouellet, D., Lizotte, M. 2007. Opening up architectures of software-intensive systems: A first prototype implementation. DRDC Valcartier TM 2006-781. Defence R&D Canada - Valcartier.

Sommaire

Pour comprendre un système logiciel orienté-objet existant, des informations concernant sa structure et son fonctionnement sont requises. Lorsqu'aucun modèle de conception complet et cohérent n'est disponible, des techniques de rétro-ingénierie telles que l'analyse statique et dynamique doivent être utilisées afin de récupérer le plus d'information possible. Bien que plusieurs outils CASE (Computer-Aided Software Engineering) offrent des capacités pour faire la rétro-ingénierie des structures statiques de systèmes orientés-objet, certains problèmes persistent et doivent être abordés. La rétro-ingénierie du fonctionnement d'un système orienté-objet est encore plus difficile que la compréhension de sa structure en raison des caractéristiques particulières liées au paradigme orienté-objet, tel que l'héritage, le polymorphisme et la liaison dynamique. Celles-ci rendent la compréhension du fonctionnement difficile en n'utilisant que l'analyse de code. Par conséquent, des modèles dynamiques doivent être extraits au moyen de l'analyse dynamique. Bien que plusieurs approches aient été proposées pour l'extraction de modèles dynamiques pour de tels systèmes, la grande quantité d'information générée dans les traces d'exécution reste un problème de taille.

Ce mémorandum technique présente un prototype, développé par les membres du projet intitulé "Ouverture d'Architectures de Systèmes Informatisés Significativement" (OASIS), qui aborde le problème d'explosion de traces en utilisant le masquage d'information. Implanté sous la forme de plugiciels dans l'Environnement de Développement Intégré (EDI) Eclipse, ce prototype permet de faire la rétro-ingénierie de modèles dynamiques à partir de systèmes logiciels Java. Ces modèles sont représentés sous la forme de diagrammes de séquence UML (Unified Modeling Language). De tels diagrammes montrent les interactions, en termes de messages ou de transferts d'information, disposées de façon séquentielle, entre les nœuds opérationnels d'un système. Le masquage d'information est accompli en reconstruisant les diagrammes à différents niveaux d'abstraction. Les interactions entre les nœuds opérationnels d'un système peuvent être affichées à un bas niveau, c.-à-d. au niveau des objets. Cependant, les nœuds opérationnels rattachés peuvent aussi être regroupés dans des structures de plus haut niveau, c.-à-d. des paquetages. À ce niveau d'abstraction, la taille des diagrammes est réduite considérablement. Les messages échangés entre les objets d'un même paquetage sont alors encapsulés dans ce dernier et ne sont pas affichés. De plus, le niveau paquetage aide à associer les fonctionnalités du système à leurs éléments architecturaux, ce qui est essentiel lors de la compréhension d'un système inconnu sur le plan de l'architecture.

À la suite de l'implantation de ce prototype, il est prévu d'en développer un second comportant un ensemble de fonctionnalités améliorées ainsi que des vues additionnelles. Cette nouvelle version sera mieux conçue pour aider les Forces canadiennes (FC) à récupérer et comprendre les architectures de leurs systèmes logiciels hérités. Idéalement, une étude qualitative devrait être menée afin d'évaluer la valeur ajoutée du prototype de récupération d'architectures OASIS sur la compréhension des participants.

Charland, P., Dessureault, D., Ouellet, D., Lizotte, M. 2007. Opening up architectures of software-intensive systems: A first prototype implementation. DRDC Valcartier TM 2006-781. R&D pour la défense Canada - Valcartier.

This page intentionally left blank.

Table of Contents

List of Figures

List of Tables

1. Introduction

Over the years, the needs of the Canadian Forces (CF) for systems interoperability have significantly increased. For example, to improve the automation of the Command, Control, Communications, Computers, Intelligence, Surveillance, and Reconnaissance (C4ISR) process, a large number of software intensive systems must interact together to handle a massive amount of information. The CF also require systems interoperability when they collaborate with allied nations to achieve common objectives.

As the CF demand greater systems interoperability, their software architects need techniques and tools to understand the architecture of existing systems and make them interoperate in order to build a system of systems (SoS). A SoS is an assemblage of components which individually may be regarded as systems and which possess two additional properties: operational and managerial independence of the components [1]. Each component system must be able to operate independently if the SoS is disassembled. Furthermore, even though the component systems are separately acquired and integrated, they maintain a continuing operating existence independent of the SoS. An example of a SoS is a system built for a coalition operation, where each participating nation brings its own operational planning system.

Before existing systems can interoperate, their architectures first need to be understood. The architecture of a system can be defined as the structure of its components, their interrelationships, as well as the principles and guidelines governing their design and evolution over time [2]. However, understanding the architecture of systems can prove to be quite a complex task. These systems have most probably undergone several code revisions without a real concern about maintaining their architectural design documentation up to date [3]. As a result, architecture recovery has to be performed to regenerate coherent abstractions and guide architects during their comprehension task. Architecture recovery can be described as the process of retrieving up-to-date architectural information from existing source code artefacts. The rational of system architectural recovery is to provide reasoning behind the software architecture or high-level organization of a system.

To support the effort of developing methodologies, techniques, and tools needed for the recovery and comprehension of existing systems' architecture, the SoS section of Defence Research and Development Canada (DRDC) Valcartier started a project called Opening up Architectures of Software-Intensive Systems (OASIS) [4]. Its objective is to develop technical solutions in order to reduce the time needed to comprehend systems to be integrated into a SoS.

In a previous phase of the OASIS project, a state-of-the-art survey of the current techniques and tools for architecture recovery and comprehension was carried out [5]. Following this survey, a qualitative study was conducted. Its objective was to assess the added value of a selected subset of the tools previously identified on the understanding of participants performing high-level comprehension tasks on large-scale military systems [6]. Using the results obtained as part of the previous two project accomplishments, a functional architecture of the ideal tool for system architecture recovery and comprehension was conceptualized [7].

The present technical memorandum describes the first OASIS prototype. This prototype implements a selected subset of the functional architecture. It was developed in Java as Eclipse [8] plug-ins. Eclipse is an extensible open source integrated development environment (IDE). The remainder of this technical memorandum is organized as follows: Section 2 presents the functionalities which were implemented as part of OASIS v1, the most important one being the generation of sequence diagrams. Section 3 and 4 respectively explains the concept of sequence diagrams in detail as well as the current approaches to reverse engineer them from existing software systems. Section 5 describes the technique developed as part of OASIS v1 to visualize sequence diagrams at the architectural level. In Section 6, the implementation of OASIS v1 is discussed. Finally, Section 7 provides conclusions and future work.

2. OASIS v1 Functionalities

The figure below shows a visual representation of the OASIS functional architecture. As indicated, it consists of the following subsystems: Repositories, Data Access, Information Management, Fact Extraction, Analysis, Synthesis, Visualization, Documentation Generation, Comprehension Process, and Graphical User Interface. For a more detailed description of each of these subsystems, please refer to [7]. In the diagram, the elements highlighted in yellow were either implemented or partially implemented as part of OASIS v1.

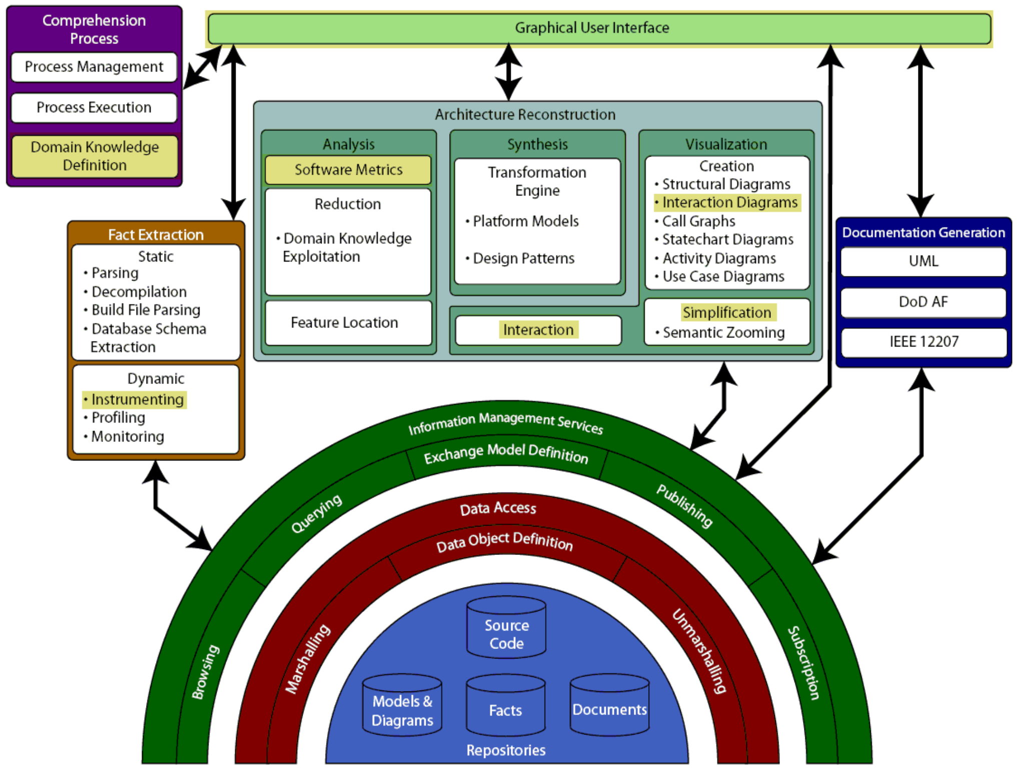
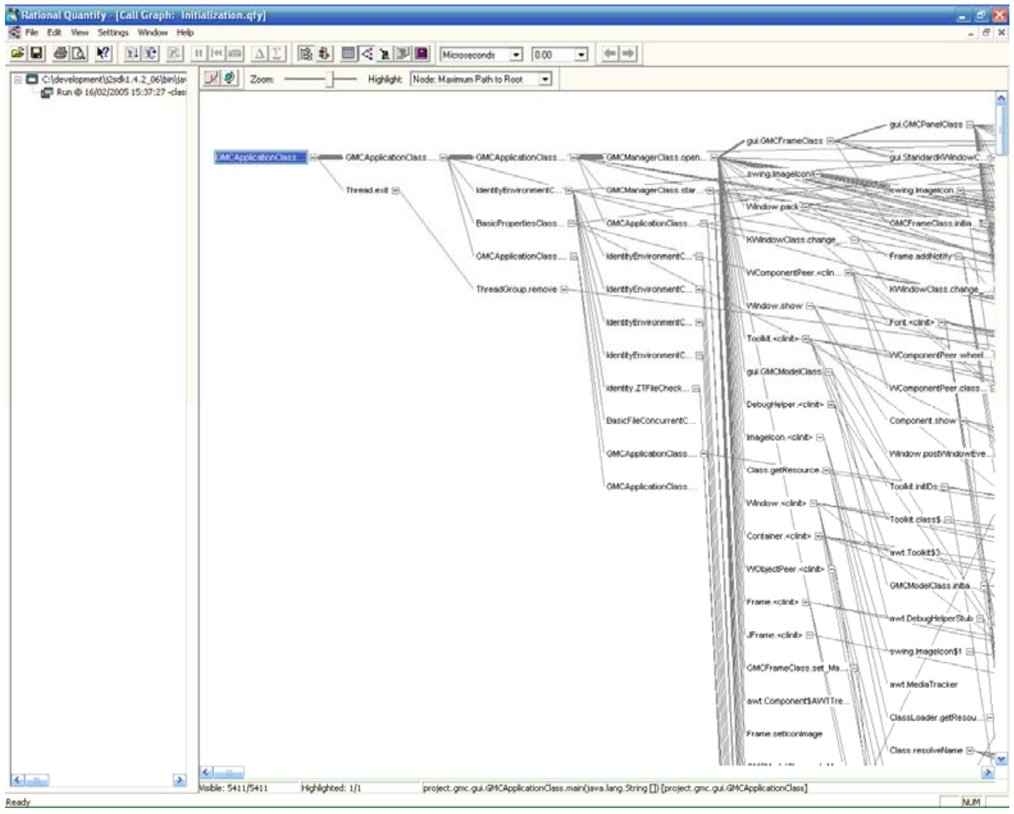


Figure 1. *OASIS Functional Architecture*

For the first implementation of the OASIS prototype, it was decided to mainly focus on the extraction, analysis, and visualization of dynamic information, i.e., information which is obtained by observing the system during execution [9]. This choice was motivated by the fact that static comprehension support through tools and techniques is a maturing research area, with the availability of a wide range of tools supporting the recovery process for a variety of systems and environments [5]. However, with an increasing part of today's legacy software systems being object-oriented and/or distributed, these tools are unable to analyze constructs such as inheritance, polymorphism, as well as dynamic binding and which result in the fact that the exact behavior of a system is only known at runtime.

Another reason why it was decided to concentrate on dynamic information for OASIS v1 is because the graphical descriptions of software architectures generated by current tools often focus on static calls and data relationships gathered by parsing the source code. These types of architecture graphs can exhibit extremely high connectivity and possess little contextual information with respect to the nature of the relationships between components [5]. Some existing tools support dynamic visualization and structure querying, but at the object level only. Therefore, the visualization they provide is hard to scale and interpret for large and distributed applications [5].

The latter findings were confirmed by the qualitative study previously conducted in [6]. One of the observations which were drawn from the results was that the biggest drawback of the selected tools was that they did not always provide the appropriate viewpoints, abstraction levels, and filters needed to understand the architecture of an application. The participants were quickly swamped by a mass of irrelevant low level details. This was especially true in the case of Rational PureCoverage and Quantify [10], the two dynamic analysis tools used as part of the study. Most of the information provided by them was at the method level. Figure 2 below shows an example where the participants were overwhelmed by the large amount of information displayed in a Rational Quantify call graph.



Visible: 5411/5411 Highlighted: 1/1 project.gmc.gui.GMCApplicationClass.main(java.lang.String []) [project.gmc.gui.GMCApplicationClass]

Figure 2. *Rational Quantify Call Graph*

The application used for the previous example was the Human Computer Interface (HCI) component of Concept Analysis and Simulation Environment for Automatic Target Tracking and Identification (CASE ATTI). CASE ATTI [11] is a multi-sensor data fusion simulation test bed used to analyze the performance of various multi-sensor data fusion architectures and algorithms for the Canadian Patrol Frigate (CPF). It was developed by the Decision Support Systems Section at DRDC Valcartier using the Java programming language. It consists of 74,000 lines of source code contained in 565 classes.

The functionalities implemented as part of the OASIS v1 prototype serve to record the execution of systems developed in Java. Once recorded, the behavior of a system can then be visualized as a sequence diagram. The reason it was decided to display the execution of systems as sequence diagrams is because they are among the crucial diagrams used during the analysis and design of object-oriented systems. They are used to identify object responsibilities and interactions associated with each use case [12]. Stated briefly, a sequence diagram shows the interactions, in terms of messages or information transfers, between the operational nodes of a system, arranged in a time sequence. However, instead of displaying these interactions at a low level only, i.e., object level, OASIS v1 allows regrouping of related operational nodes into higher level structures, i.e., packages. This reduces the cognitive burden and, as a result, improves the software comprehension process, since it allows information to be filtered out. Therefore, a person trying to understand a system has less information to search through and can concentrate on the system's high level structures as well as the interactions between them. This feature of OASIS v1 is particularly useful in the case of large-scale systems.

In addition to the functionalities presented above, OASIS v1 provides a suite of metrics which allows analyzing a software system for the recovery and comprehension of its architecture. Also, it integrates a model driven development tool for the reverse engineering of structure diagrams such as UML class and package diagrams. Furthermore, it offers the functionality to define the domain model of a software system. This was believed important to implement, as the human-oriented recognition process depends heavily on an *a priori* contextual knowledge of the application domain, its entities, and their relationships. Using this functionality, OASIS v1 allows users to map this vocabulary of terms to an execution trace to determine which entities were executed by a particular run of the system and to map source code elements to their corresponding concept of the application domain.

3. UML Sequence Diagrams

The main functionality of OASIS v1 is the generation of sequence diagrams based on execution traces. This section details the concept of a sequence diagram.

The Unified Modeling Language (UML) is a graphical language to visualize, specify, construct, and document the artifacts of a software-intensive system [13]. In UML, one way to model the dynamic aspects of a system is to use sequence diagrams. A sequence diagram shows a set of objects and the messages sent and received by those objects. The objects are typically named or anonymous instances of classes, but can also represent instances of other things (e.g., collaborations, components, and nodes) [13]. As shown in Figure 3 on the next page, sequence diagrams emphasize the time ordering of messages.

Sequence diagrams also model the lifelines of objects. An object's lifeline, depicted as a vertical dashed line, represents the existence of an object over a period of time. The objects that will be in existence for the whole duration of the interaction are aligned at the top. Their lifelines are drawn from the top to the bottom of the diagram. Objects can also be created and destroyed during an interaction. In the former case, their lifelines start with the receipt of the message stereotyped as `create`. In the latter case, their lifelines end with the receipt of the message stereotyped as `destroy`, and are given the visual cue of a large X, marking the end of their lives [13].

Another concept present in sequence diagrams is the focus of control. The focus of control is a thin rectangle that shows the period of time during which an object is performing an action, either directly or through a subordinate procedure [13]. The top of the rectangle is aligned with the start of the action and the bottom, with its completion. The latter can also be marked by a return message. The nesting of focus of control caused either by recursion, a call to a self-operation, or by the call-back from another object, is represented by stacking another focus of control slightly to the right of its parent. This can be repeated to an arbitrary depth.

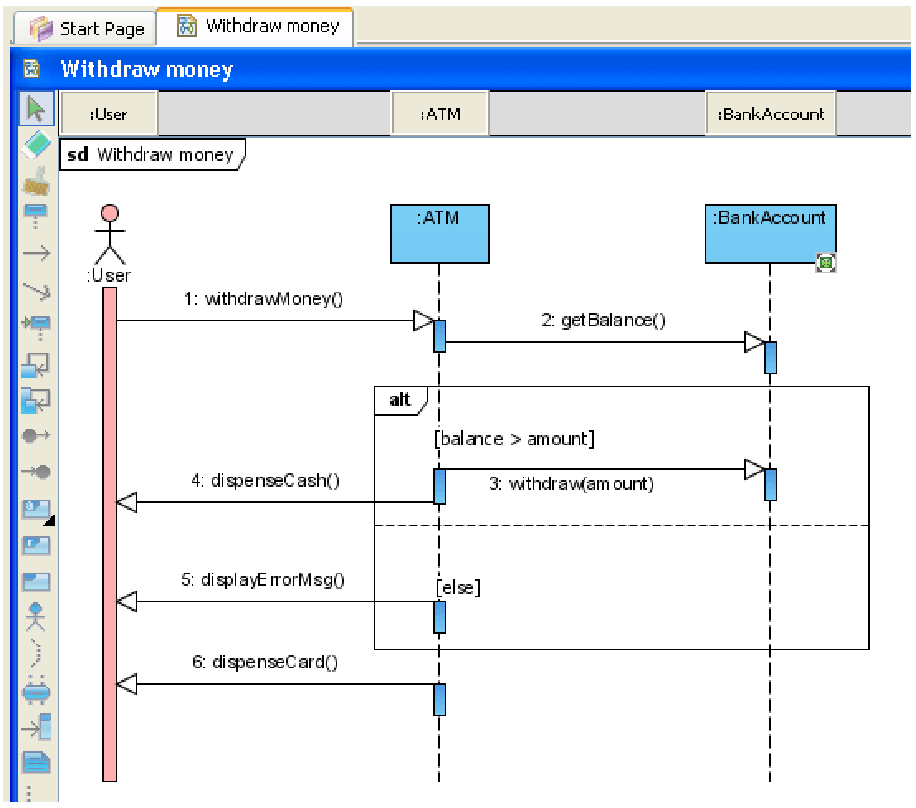


Figure 3. *UML Sequence Diagram [14]*

As previously mentioned, sequence diagrams introduce all the objects which work together to fulfill an action, as well as the messages dispatched from one object to another. When a message is passed, the resulting action is an executable statement, which forms an abstraction of a computational procedure [13]. In UML, the following kinds of actions can be modeled:

Table 1. *Kinds of Actions in UML [13]*

ACTION	DESCRIPTION
Call	Invokes an operation on an object. An object can send a message to itself, resulting in the local invocation of an operation.
Return	Returns the value to the caller.
Send	Sends a signal to an object.
Create	Creates an object.
Destroy	Destroys an object.

Figure 4 below provides a visual distinction among the different kinds of UML messages. The most common kind of messages modeled in sequence diagrams is the call, where one object invokes an operation of another or same object.

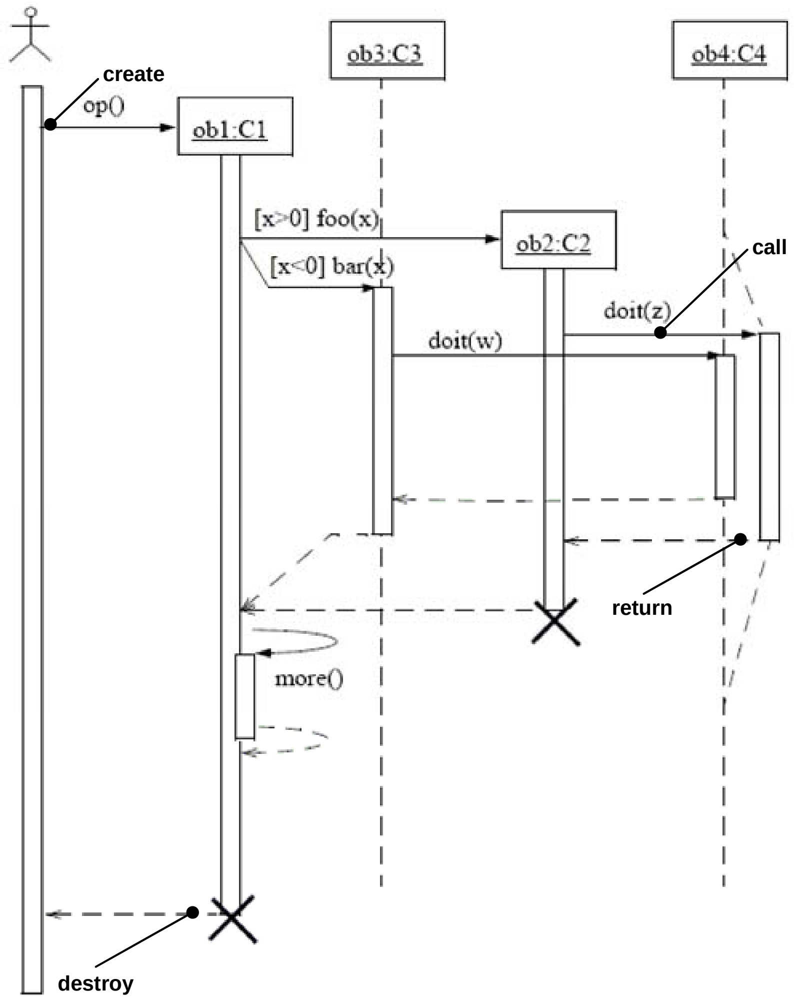


Figure 4. Kinds of Messages in UML [15]

4. Reverse Engineering of Sequence Diagrams

UML is the *de facto* standard for object-oriented software development [16]. Its models can be directly connected to a variety of programming languages such as C++ and Java [13]. This mapping allows forward engineering, i.e., the generation of code from a UML model into a programming language. However, the reverse is also possible: a model can be reconstructed from an implementation back into UML. This is especially useful when the source code of a software system is the only documentation available, as in the case of the systems targeted by the OASIS research project. In these circumstances, to completely understand the legacy system, information regarding its structure, behavior, and internal states has to be extracted from the source code or its execution, and must be represented in the form of an abstract model such as UML. This process of analyzing the source code of a software system to represent it at a higher level of abstraction, by extracting architectural artifacts from the code, is called reverse engineering [16].

Reverse engineering sequence diagrams is possible. This section presents an overview of some of the existing tools available for the reverse engineering of Java source code back to sequence diagrams. These tools are either based on static or dynamic analysis of the system under study.

4.1 Static Analysis

Static analysis is performed by analyzing the source code of a software system without executing it [9]. In this section, two tools which perform static reverse engineering of sequence diagrams are presented: EclipseUML [17] and the Sequence Diagram Viewer NetBeans module [18]. Note that there exist other tools which can statically reverse engineer sequence diagrams such as Borland Together [19] and the NetBeans UML Modeling module [20]. The reason those are not presented in this document is because their functionalities are very similar to the ones of the tools covered.

4.1.1 Omondo EclipseUML

EclipseUML Studio is an Eclipse plug-in developed by the Omondo company. It is a model driven development solution based on UML and the Eclipse Modeling Framework (EMF). It also offers reverse engineering functionalities. Among them is the capacity to reverse engineer the sequence diagram of a method, as illustrated in Figure 5. This example shows the corresponding sequence diagram for the `main` method of JUnit's `TestRunner` class displayed in Figure 6. JUnit [21] is a Java framework for unit testing.

The major limitation of sequence diagrams reverse engineered by EclipseUML is that they do not model the children calls of the selected method. For example, in Figure 5, it is not shown that the constructor of the class `BaseTestRunner` is called by the constructor of the class `TestRunner`. Another limitation is that the generated sequence diagrams cannot be visualized at an abstract level higher than the object level.

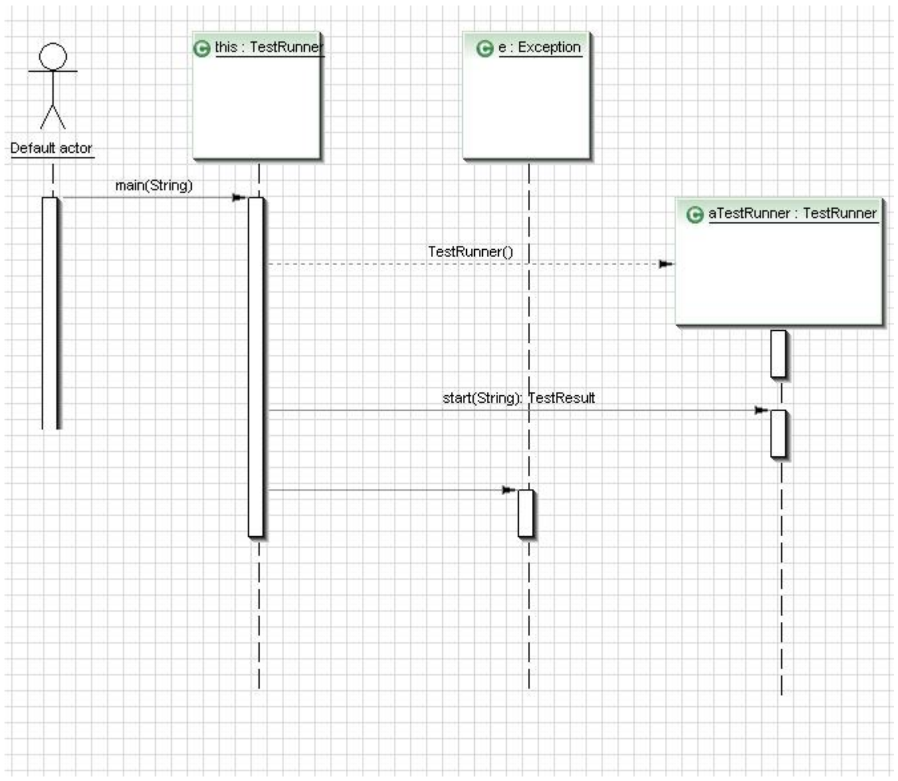
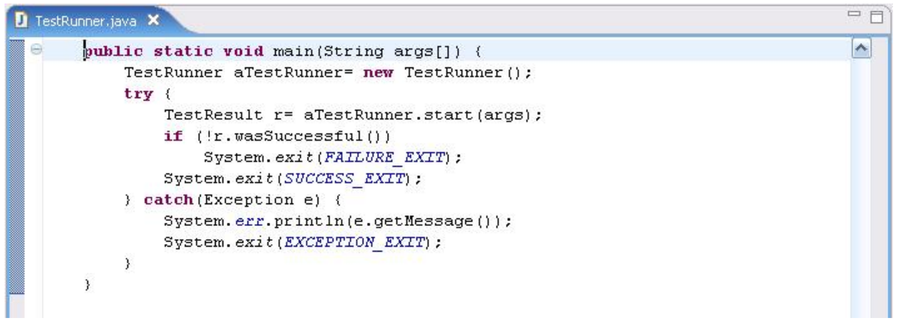


Figure 5. *Sequence Diagram Generated by EclipseUML*



```
public static void main(String args[]) {
    TestRunner aTestRunner= new TestRunner();
    try {
        TestResult r= aTestRunner.start(args);
        if (!r.wasSuccessful())
            System.exit(FAILURE_EXIT);
        System.exit(SUCCESS_EXIT);
    } catch(Exception e) {
        System.err.println(e.getMessage());
        System.exit(EXCEPTION_EXIT);
    }
}
```

Figure 6. *Source Code of the main Method of the JUnit TestRunner Class*

4.1.2 Sequence Diagram Viewer NetBeans Module

NetBeans [22] is an open source IDE supporting the development of Java applications. It can be extended by adding modules to it. One such module is the Sequence Diagram Viewer [18], which generates a sequence diagram from a selected constructor or method. Figure 7 below displays a section of the sequence diagram produced by the Sequence Diagram Viewer module for the `main` method of the JUnit `TestRunner` class.

The advantage that the Sequence Diagram Viewer has over EclipseUML is that the sequence diagrams constructed start from the selected method and model all the children calls recursively. For example, in Figure 7, the object of type `TestRunner` creates an object of type `ResultPrinter`. This was not modeled in the previous sequence diagram generated by EclipseUML.

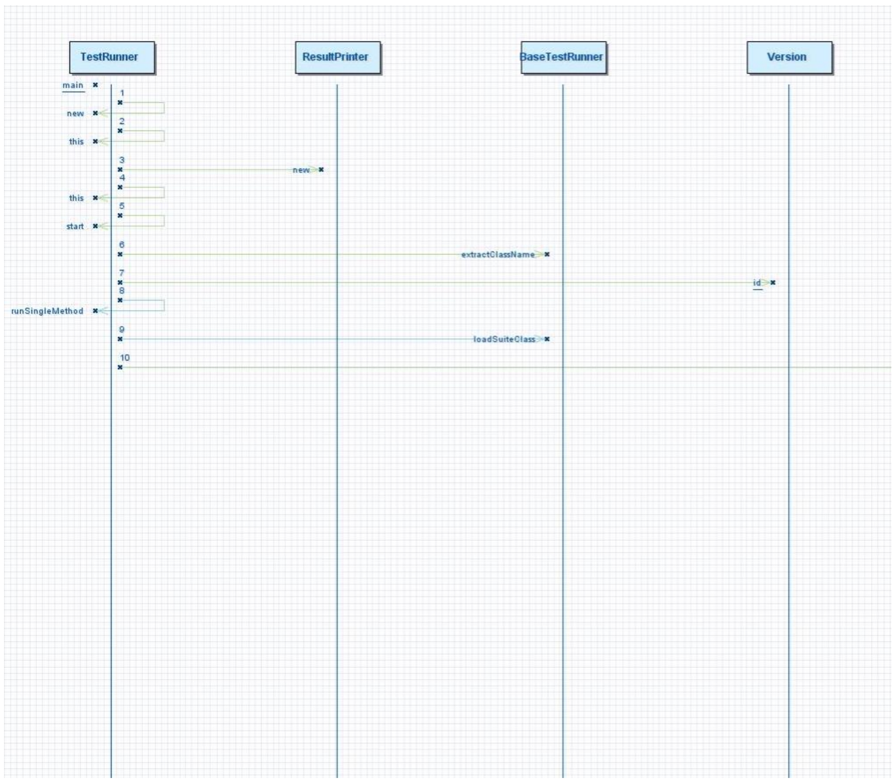


Figure 7. *Sequence Diagram Generated by the Sequence Diagram Viewer*

4.1.3 Limitations of Static Analysis

Reverse engineering sequence diagrams from source code is difficult. One of the main reasons is that because of inheritance, polymorphism, and dynamic binding, it is difficult and sometimes even impossible to know, using only the source code, the dynamic type of an object and therefore, which method is going to be executed. Multithreading and distribution further complicate the analysis [23]. As a result, the generation of sequence diagrams is difficult, as identifying method call sequences from source code requires complex techniques, such as symbolic execution, in addition to source code analysis, which are not applicable in the case of large and complex systems [24].

Although static analysis can represent a complete picture of what could happen at runtime, it does not necessarily show what actually happens [23]. Therefore, if one wants to produce meaningful sequence diagrams from existing large and complex software systems, their executions have to be monitored, as the exactness of the generated sequence diagrams depends extensively on runtime behavior.

4.2 Dynamic Analysis

Dynamic analysis is the process of analyzing the behavior of a software system during its execution [9]. The objective is to increase the level of precision provided by static analysis and as a result, improve the comprehension of the software system under study. More precisely, the purpose of dynamic analysis is to record the effective flow of control, i.e., the sequence of interactions, of a system execution [25]. This section presents two tools which can reverse engineer dynamic sequence diagrams of Java software systems. These are J2U [26] and the Eclipse TPTP project [27].

4.2.1 J2U

Java to UML (J2U) is a tool developed by the NASRA company which allows reverse engineering sequence diagrams of software systems based on an execution trace. An execution trace is a record of the sequence of instructions executed that often takes the form of a list of code labels encountered [28]. Figure 8 on the next page shows a portion of a sequence diagram reverse engineered by J2U using JUnit.

Because the sequence diagrams generated by J2U are based on an execution trace, they should accurately reflect the behavior of the software system under study. Unfortunately, this is not the case. This is due to the fact that J2U does not model constructor calls. As a result, the generated sequence diagrams do not exactly correspond to the actual sequence of method calls of the underlying system.

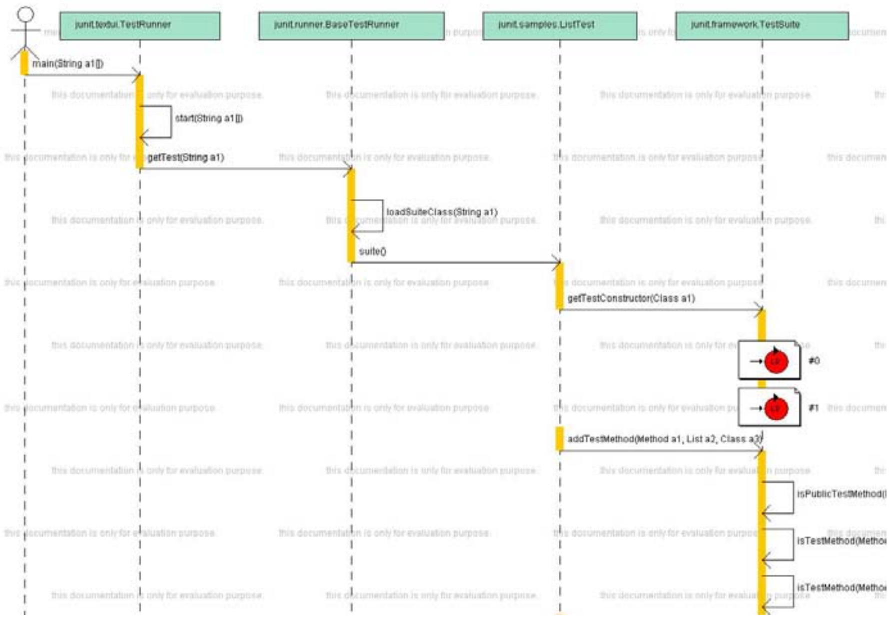

junit.textui.TestRunner
junit.runner.BaseTestRunner
junit.samples.ListTest
junit.framework.TestSuite
main(String a1[])
start(String a1[])
getTest(String a1)
loadSuiteClass(String a1)
suite()
getTestConstructor(Class a1)
#0
#1
addTestMethod(Method a1, List a2, Class a3)
isPublicTestMethod(
isTestMethod(Metho
isTestMethod(Metho
this documentation is only for evaluation purpose.

Figure 8. *Sequence Diagram Generated by J2U*

4.2.2 The Eclipse TPTP Project

The Eclipse Test and Performance Tools Platform (TPTP) project is an open platform supplying frameworks and services to build test and performance tools that can be integrated with the Eclipse platform and other tools. The TPTP profiling tool enables to pinpoint performance and memory usage problems within applications using visualization features. Among the views it offers is the UML2 Trace Interaction view, which presents the execution flow of a software system in the form of a sequence diagram, as illustrated in Figure 9.

The sequence diagrams can be viewed at different levels of abstraction, starting with object interactions, through interactions among threads or processes, up to hosts interactions across a network. Furthermore, lifelines and messages can be collapsed and expanded to hide or expand sections of data within a view. For example, in Figure 10, the second lifeline in Figure 9 (`Laucher$Ap…`) as well as the fifth message (`Properties`) were collapsed.

Although the sequence diagrams reverse engineered by the Eclipse TPTP project can be viewed at different levels of abstraction, there is too much discrepancy between the granularity levels it offers. For example, the sequence diagrams at the object level may expose a user with too much low level information, while the view at the thread or process level may not provide enough information to understand a software system one is not familiar with.

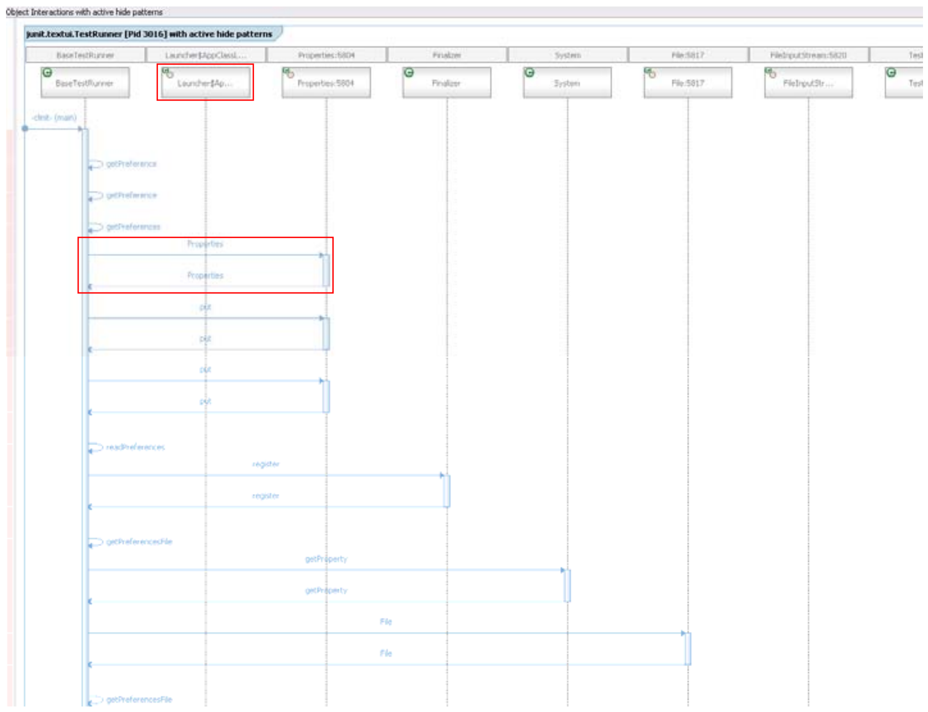


Figure 9. *Sequence Diagram Generated by the Eclipse TPTP Project*

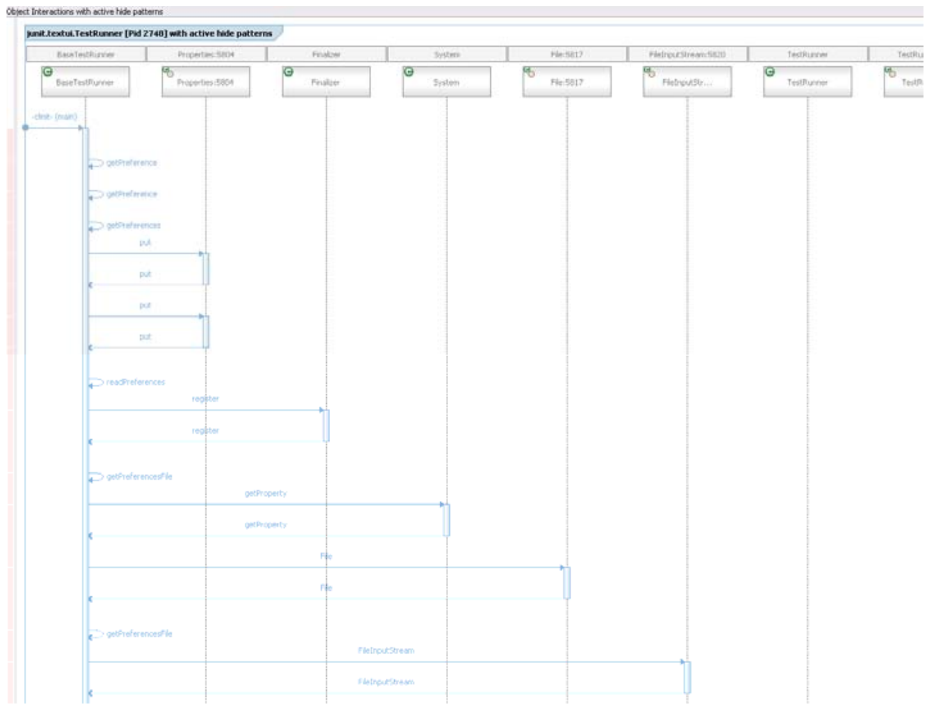


Figure 10. *Sequence Diagram with Lifelines and Messages Collapsed*

4.2.3 Limitations of Dynamic Analysis

Although dynamic analysis supports the specific characteristics of the object-oriented paradigm such as dynamic binding and polymorphism, one of the problems faced when dynamically analyzing object-oriented software systems is the volume of information generated by execution traces [29]. This could be a problem for the OASIS project, since the systems to be analyzed will be large-scale military applications developed in Java and consisting of more than 1,000 classes. In order to reduce the volume of information, existing approaches apply techniques such as filtering, pattern matching, sampling, and information hiding [30]. The approach used by OASIS v1 to deal with the trace explosion problem is information hiding. This is achieved by visualizing the generated sequence diagrams at various levels of abstraction. Execution traces can be visualized at the object level, as it is the case in standard UML sequence diagrams. However, they can also be visualized at the package level, therefore hiding the messages exchanged among the classes of the same package.

5. Extraction of Dynamic Views in OASIS v1

A dynamic analysis aimed at reverse engineering sequence diagrams must address three different but complementary issues. First, an instrumentation strategy has to be devised to collect at runtime the information necessary to generate complete and correct sequence diagrams at the level of details needed [23]. The impact of instrumentation on the execution of the software system should be reduced to the maximum possible extent. The second issue to consider is the definition of a logging strategy to store, in an appropriate format, the data produced when executing the instrumented software system [23]. Finally, a visualization technique must be devised to effectively display the reverse engineered sequence diagrams [23].

Figure 11 below illustrates the high-level strategy used in OASIS v1 for the reverse engineering of sequence diagrams based on execution traces. First, the Java bytecode of the software system under study is instrumented. The instrumented bytecode is then executed by the Java Runtime Environment (JRE) and the resulting execution data is logged in an execution trace (.xrat file). Next, the execution trace is read and displayed as a sequence diagram in an Eclipse view.

The remaining of this section explains each of these steps in more detail. The aspects related to their implementation are covered in Section 6.

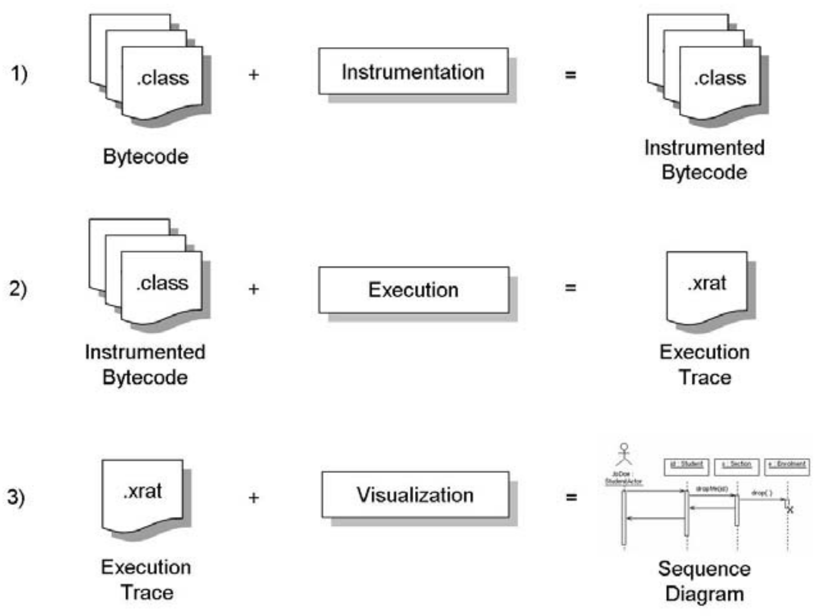


Figure 11. *Extraction of Dynamic Views in OASIS v1*

5.1 Instrumentation

There are four different alternatives to obtain runtime information about a software system: code instrumentation, annotation of runtime environments, post mortem analysis, and on-line debugging or profiling [31]. The alternative which was considered for the first implementation of the OASIS prototype is instrumentation. This approach was favored over the others because of its low performance impact and the fact that its implementation is not inherently complex and does not require huge efforts, compared to annotating runtime environments and on-line debugging.

As opposed to general-purpose program transformations, instrumentation only aims to gather additional information about a system, rather than modify its original structure and behavior, allowing only minor side effects, such as increases in execution time or changes to the log file [32]. As an example, Java bytecode instrumentation uses structural and semantic information provided by the language and platform specifications to both identify instrumentation points as well as avoid affecting the original program structure and behavior [32]. Such instrumentation does not remove program elements (e.g., classes, fields, and methods). Variables defined by the original program may be read but not written. Instrumentation may add its own variables, even to existing program elements (e.g., new fields or local variables), and those variables may be read or written by it. Instrumentation may also insert new code into original program methods, and invoke other methods from this code, provided that original variables are not modified as a result of these invocations. Finally, instrumentation may outline code, i.e., move all or part of the method code into a new method and replace it in the original method with the invocation of the new one [32]. Once executed, an instrumented software system generates an execution trace.

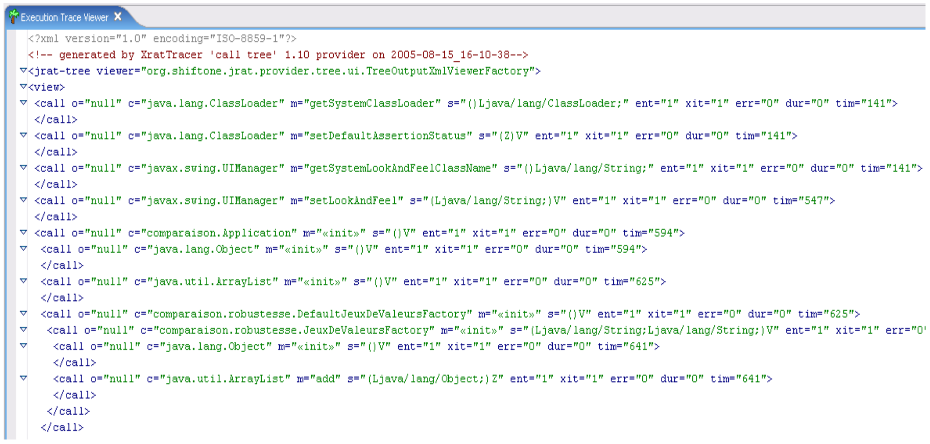
There are two different kinds of instrumentation: source and binary. In the first case, the source code is parsed and statements are added to retrieve the required information at runtime. In the second one, trace statements are inserted into the bytecode, which includes applications as well as dynamic and shared libraries. Instrumenting source code is easier than bytecode, as one can work in a high-level language. However, the disadvantage is that after it has been instrumented, the modified source code has to be recompiled in order to be able to execute the tracing statements and therefore, extract dynamic information.

Due to the additional overhead for recompiling instrumented source code and the fact that the objective of the OASIS project is to recover and comprehend the architecture of large scale military software systems consisting of more than 1,000 classes, bytecode instrumentation was selected. This choice was motivated by the fact that it is not intrusive in the source code and it allows to specify (1) the types of entities to instrument, (2) the parts of the code in which those entities must be instrumented, and (3) the kind of information to collect from the different entity types [33].

One limitation of instrumentation is that the behavior of the instrumented system may be different from the expected one (e.g., deadlines), as a consequence of the delays introduced by the execution of the added code [23]. This issue is unavoidable, as observing a system changes the system [34]. However, this should not be a problem in the present case, as the systems targeted by the OASIS research project are not, at the moment, hard real-time systems with deadlines. As a result, the delays introduced by the instrumentation should not change the intended behavior of the system. Also, in order to limit the impact of instrumentation, only the constructs required to obtain the necessary information are instrumented.

5.2 Logging Strategy

As previously illustrated in Figure 11, once the software system under study has been instrumented, it then has to be executed. In OASIS v1, when the statements inserted into the bytecode during instrumentation are executed, they produce trace statements, i.e., text lines, in the trace file. This trace file contains methods entry and exit, along with the method signature and the class of the target object, i.e., the object executing the method. Also, a timestamp based on each method local time is inserted to indicate when the method execution occurred. In the trace file, methods are ordered by their execution sequence and indented according to their calling hierarchy. Figure 12 below shows an example of a log file generated by OASIS v1.



```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- generated by XratTracer 'call tree' 1.10 provider on 2005-08-15_16-10-38-->
<jrat-tree viewer="org.shiftone.jrat.provider.tree.ui.TreeOutputXmlViewerFactory">
<view>
 <call o="null" c="java.lang.ClassLoader" m="getSystemClassLoader" s="()Ljava/lang/ClassLoader;" ent="1" xit="1" err="0" dur="0" tim="141">
 </call>
 <call o="null" c="java.lang.ClassLoader" m="setDefaultAssertionStatus" s="(Z)V" ent="1" xit="1" err="0" dur="0" tim="141">
 </call>
 <call o="null" c="javax.swing.UIManager" m="getSystemLookAndFeelClassName" s="()Ljava/lang/String;" ent="1" xit="1" err="0" dur="0" tim="141">
 </call>
 <call o="null" c="javax.swing.UIManager" m="setLookAndFeel" s="(Ljava/lang/String;)V" ent="1" xit="1" err="0" dur="0" tim="547">
 </call>
 <call o="null" c="comparaison.Application" m="«init»" s="()V" ent="1" xit="1" err="0" dur="0" tim="594">
  <call o="null" c="java.lang.Object" m="«init»" s="()V" ent="1" xit="1" err="0" dur="0" tim="594">
  </call>
  <call o="null" c="java.util.ArrayList" m="«init»" s="()V" ent="1" xit="1" err="0" dur="0" tim="625">
  </call>
  <call o="null" c="comparaison.robustesse.DefaultJeuxDeValeursFactory" m="«init»" s="()V" ent="1" xit="1" err="0" dur="0" tim="625">
   <call o="null" c="comparaison.robustesse.JeuxDeValeursFactory" m="«init»" s="(Ljava/lang/String;Ljava/lang/String;)V" ent="1" xit="1" err="0'
    <call o="null" c="java.lang.Object" m="«init»" s="()V" ent="1" xit="1" err="0" dur="0" tim="641">
    </call>
    <call o="null" c="java.util.ArrayList" m="add" s="(Ljava/lang/Object;)Z" ent="1" xit="1" err="0" dur="0" tim="641">
    </call>
   </call>
  </call>
```

Figure 12. *Sample Log File*

5.3 Visualization

Once the execution trace file has been created, OASIS v1 reads it and generates the corresponding sequence diagram in an Eclipse view, as illustrated in Figure 13 below.

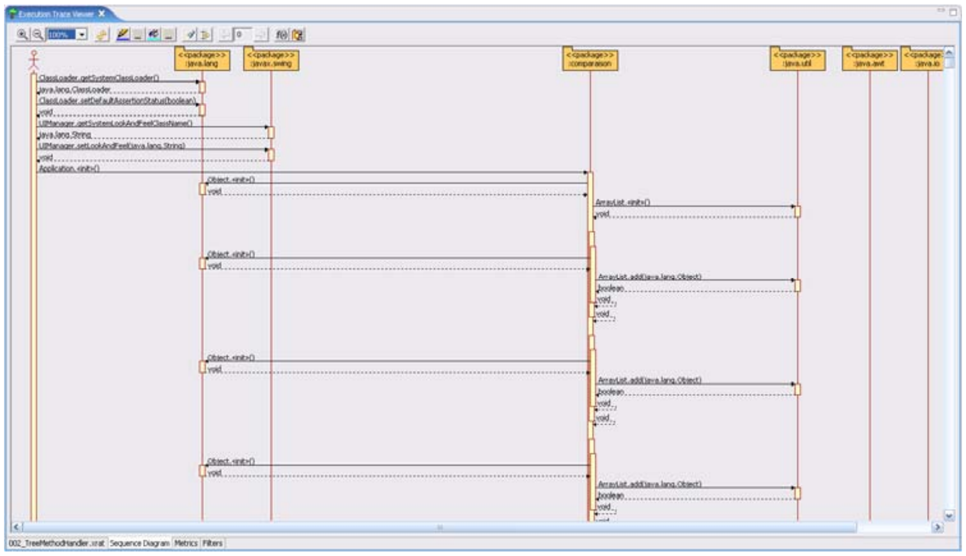


Figure 13. *Sequence Diagram Generated by OASIS v1*

As mentioned in Section 4.2 of the present document, there already exist tools which can generate sequence diagrams based on system execution. However, they suffer from the trace explosion problem, i.e., they expose users to a large volume of information. The Eclipse TPTP project attempts to deal with this problem by offering to view the sequence of execution flow from different abstraction levels, starting with class interactions, through interactions among threads or processes, up to hosts interactions across a network. Although the thread and process views allow a considerable reduction of the diagrams size, they are not useful to map system functionalities to architectural elements, something which is essential when trying to understand an unfamiliar system at the architectural level.

The approach used in OASIS v1 to address the trace explosion problem is information hiding. As in the Eclipse TPTP project, information hiding is achieved by displaying the sequence diagrams at different abstraction levels. However, the difference between the additional degree of abstraction provided by OASIS v1 is that it offers a higher degree of generalization than the Eclipse TPTP class view, while being at a lower level than the thread, process, and host views.

5.3.1 Object Level Sequence Diagrams

At the object level, the sequence diagrams generated by OASIS v1 display the interactions among the objects which participate in the execution of the software system. Figure 14 shows an example of such a diagram. One problem with this level of abstraction is that when used to visualize the execution trace of large-scale applications consisting of more than 1,000 classes, it can generate a very large volume of information. To address this limitation, another abstraction level is proposed.

5.3.2 Package Level Sequence Diagrams

The package level allows a considerable reduction in the diagrams size, since messages exchanged between objects of the same package are encapsulated in the latter and not shown in the diagram. Therefore, users can get the mapping from system functionalities to architectural elements. Figure 15 on the next page shows an example of a sequence diagram displayed at the package level. This abstraction level reduces the cognitive burden and, as a result, improves the software comprehension process, since it allows information to be filtered out. Therefore, a person trying to understand a system has less information to search through and can concentrate on the system's high level structures as well as the interactions between them. This abstraction level also addresses the limitation of the object level and can be used to visualize the interactions of an object-oriented software system consisting of more than 1,000 classes.

To group together all the objects which belong to the same package, the user has to right click on an object and select `Fold package classes` from the contextual menu, where `package` represents the fully qualified name of the package to which the selected object belongs. A package can also be expanded back to the object level. The user has simply to right click on it and select `Unfold package` from the contextual menu.

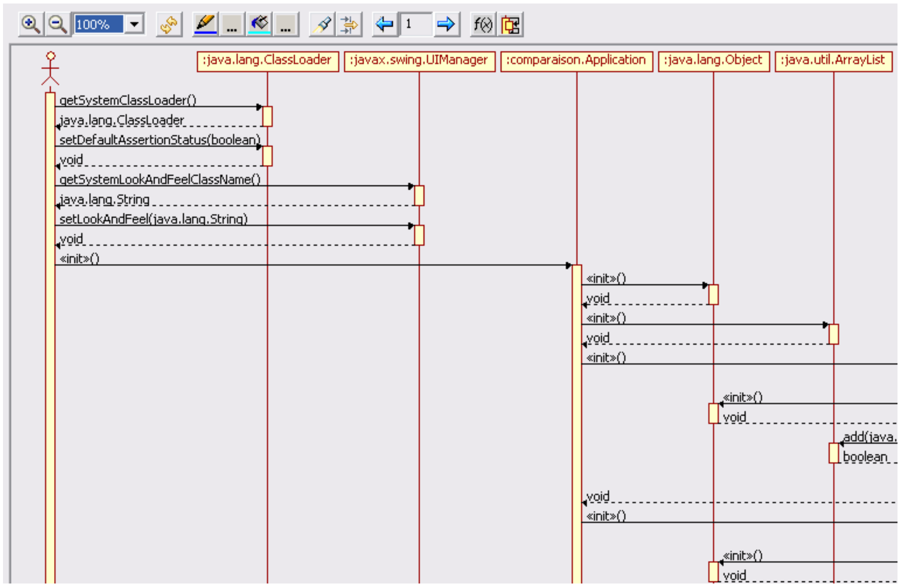
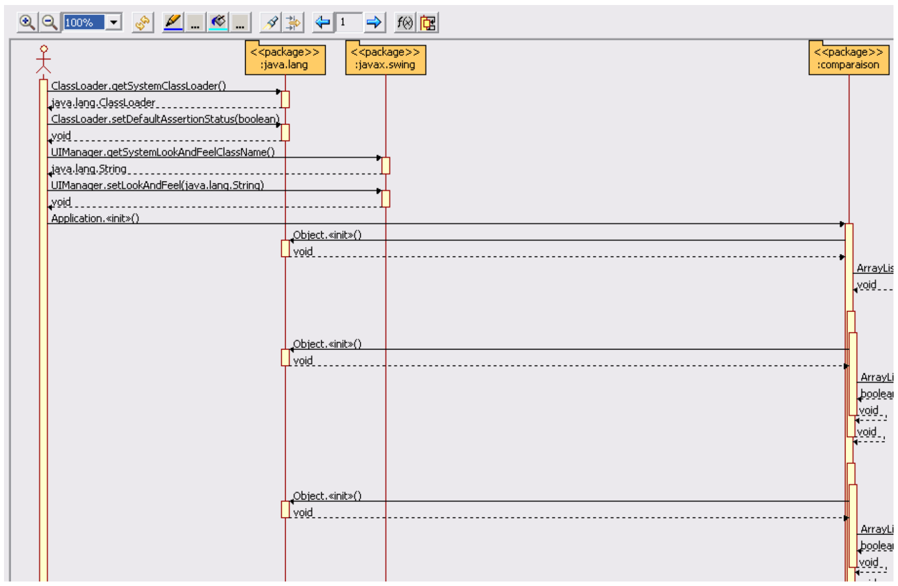


Figure 14. *Object Level Sequence Diagram*



100%
<<package>> :java.lang
<<package>> :javax.swing
<<package>> :comparaison
ClassLoader.getSystemClassLoader()
java.lang.ClassLoader
ClassLoader.setDefaultAssertionStatus(boolean)
void
UIManager.getSystemLookAndFeelClassName()
java.lang.String
UIManager.setLookAndFeel(java.lang.String)
void
Application.<init>()
Object.<init>()
void

Figure 15. Package Level Sequence Diagram

5.3.3 Method Collapsing

Another feature present in OASIS v1 to hide sections of data within the view and therefore, reduce the complexity of the generated sequence diagram, is the possibility to collapse the objects focus of control. For example, in Figure 17, the focus of control of the `comparaison` package, surrounded by a red rectangle, was collapsed for the `Application.<init>` method call. Compared to the same diagram in Figure 16, the one with the focus of control collapsed is slightly less complex, as all the methods calls invoked from the selected method are hidden. This feature is useful when one understands a method call and wants to collapse it in order to reduce the complexity of the sequence diagram.

To collapse the focus of control of a method call, the user has to right click on it to bring up the context menu and select `Fold method()`, where `method` is the fully qualified name of the method called. The collapsed message is indicated by the word `<<nested>>` being inserted before the method name. The focus of control of a method can also be expanded back by right clicking on the collapsed focus of control and selecting `Unfold method()` from the contextual menu.

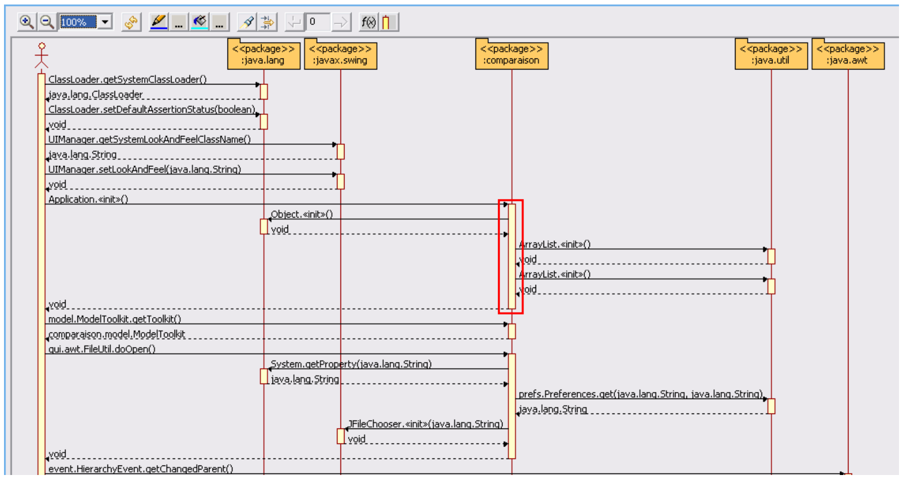

<<package>> :java.lang
<<package>> :javax.swing
<<package>> :comparaison
<<package>> :java.util
<<package>> :java.awt
ClassLoader.getSystemClassLoader()
java.lang.ClassLoader
ClassLoader.setDefaultAssertionStatus(boolean)
void
UIManager.getSystemLookAndFeelClassName()
java.lang.String
UIManager.setLookAndFeel(java.lang.String)
void
Application.«init»()
Object.«init»()
void
ArrayList.«init»()
void
ArrayList.«init»()
void
void
model.ModelToolkit.getToolkit()
comparaison.model.ModelToolkit
gui.awt.FileUtil.doOpen()
System.getProperty(java.lang.String)
java.lang.String
prefs.Preferences.get(java.lang.String, java.lang.String)
java.lang.String
JFileChooser.«init»(java.lang.String)
void
void
event.HierarchyEvent.getChangedParent()

Figure 16. *Sequence Diagram with Method Expanded*

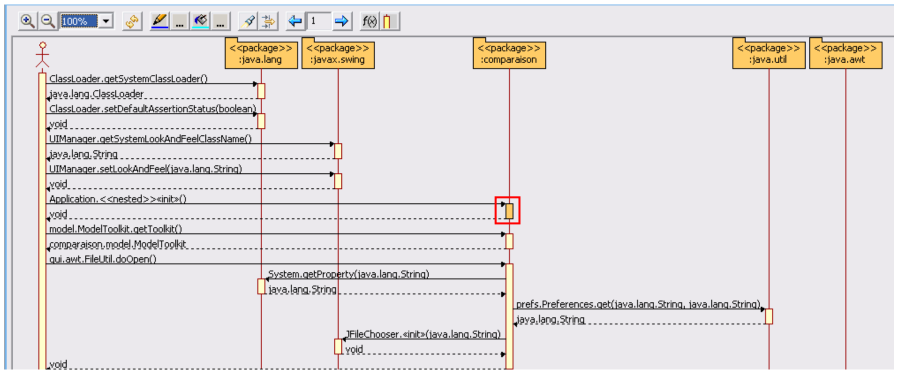
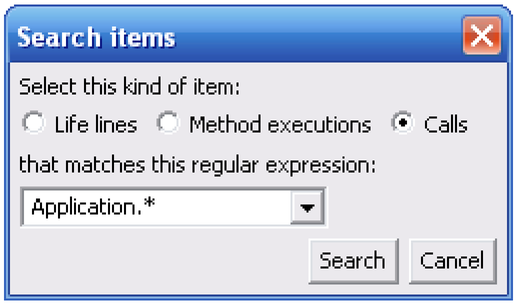

<<package>> :java.lang
<<package>> :javax.swing
<<package>> :comparaison
<<package>> :java.util
<<package>> :java.awt
ClassLoader.getSystemClassLoader()
java.lang.ClassLoader
ClassLoader.setDefaultAssertionStatus(boolean)
void
UIManager.getSystemLookAndFeelClassName()
java.lang.String
UIManager.setLookAndFeel(java.lang.String)
void
Application.<<nested>>«init»()
void
model.ModelToolkit.getToolkit()
comparaison.model.ModelToolkit
gui.awt.FileUtil.doOpen()
System.getProperty(java.lang.String)
java.lang.String
prefs.Preferences.get(java.lang.String, java.lang.String)
java.lang.String
JFileChooser.«init»(java.lang.String)
void
void

Figure 17. *Sequence Diagram with Method Collapsed*

5.3.4 Searching

The searching functionalities offered in OASIS v1 allow to find, in the sequence diagram view, objects, packages, method calls, and objects focus of control matching a regular expression. Figure 18 on the next page shows the Search dialog box.


Search items

Select this kind of item:

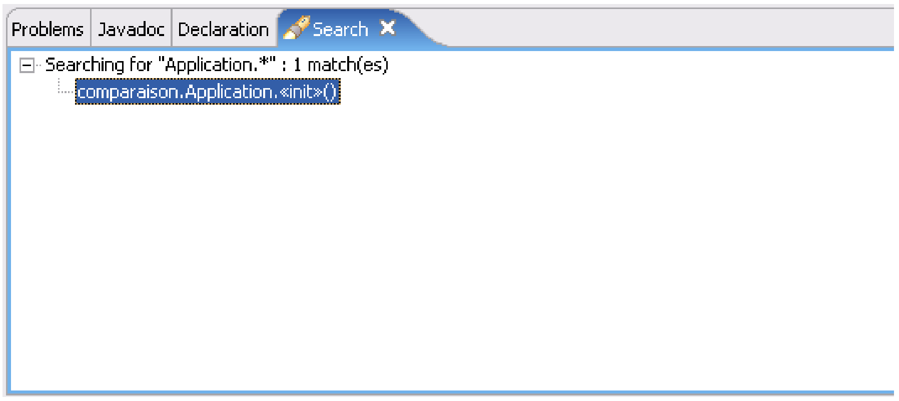
Life lines Method executions [✓ Calls]

that matches this regular expression:

Application.*

Figure 18. *Search Dialog Box*

The user first selects what to search for, i.e., life lines (objects and packages), method executions (objects focus of control), and calls (method calls). The user then specifies, in the input field, the pattern of the regular expression to search for in the sequence diagram. After having clicked on the `Search` button, the results found are shown in the Eclipse Search view, as illustrated in Figure 19.



Searching for "Application.*" : 1 match(es)

comparaison.Application.«init»()

Figure 19. *Eclipse Search View*

Clicking on a result in the list will refresh the sequence diagram view with the object, package, method call, or object focus of control matching the search criteria highlighted, as illustrated in Figure 20 on the next page.

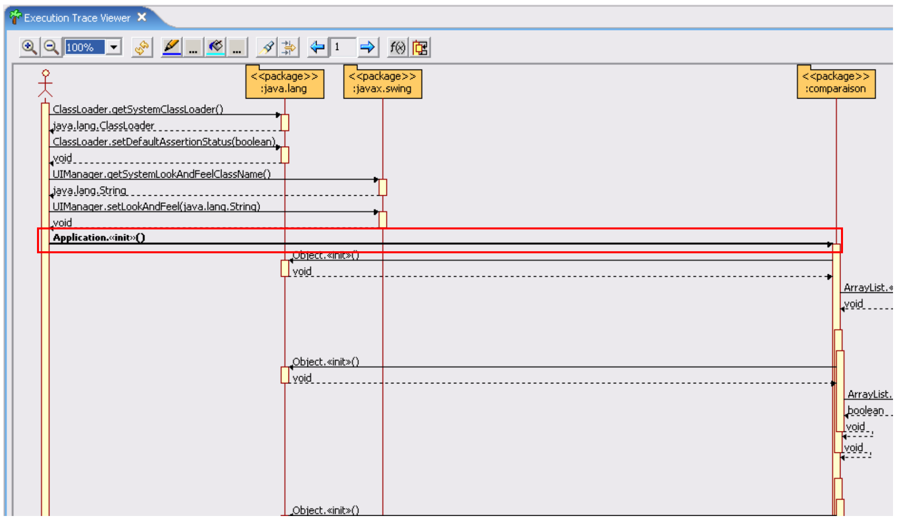


Figure 20. Sequence Diagram with a Method Call Highlighted

5.3.5 Filtering

In addition to its searching functionalities, OASIS v1 provides a filtering mechanism that can be used to reduce the amount of data displayed in the sequence diagram view. The Filter dialog box is identical to the Search dialog box. The user specifies what to filter out, i.e., objects and packages, objects focus of control, or method calls, as well as the pattern of the regular expression to match. After clicking on the `Filter Out` button, the sequence diagram view is refreshed with the elements matching the regular expression removed.

On the sequence diagram view toolbar, the following two icons offer additional filtering functionalities. They respectively allow to filter out methods signature as well as nested focus of control. Figure 21 on the next page displays the interactions between the `comparaison` and `java.util` packages of an application. Figure 22 shows the interactions between the same two packages, but with the methods signature and nested focus of control having been filtered out. As one can see, the amount of data displayed in the second sequence diagram view has been significantly reduced.

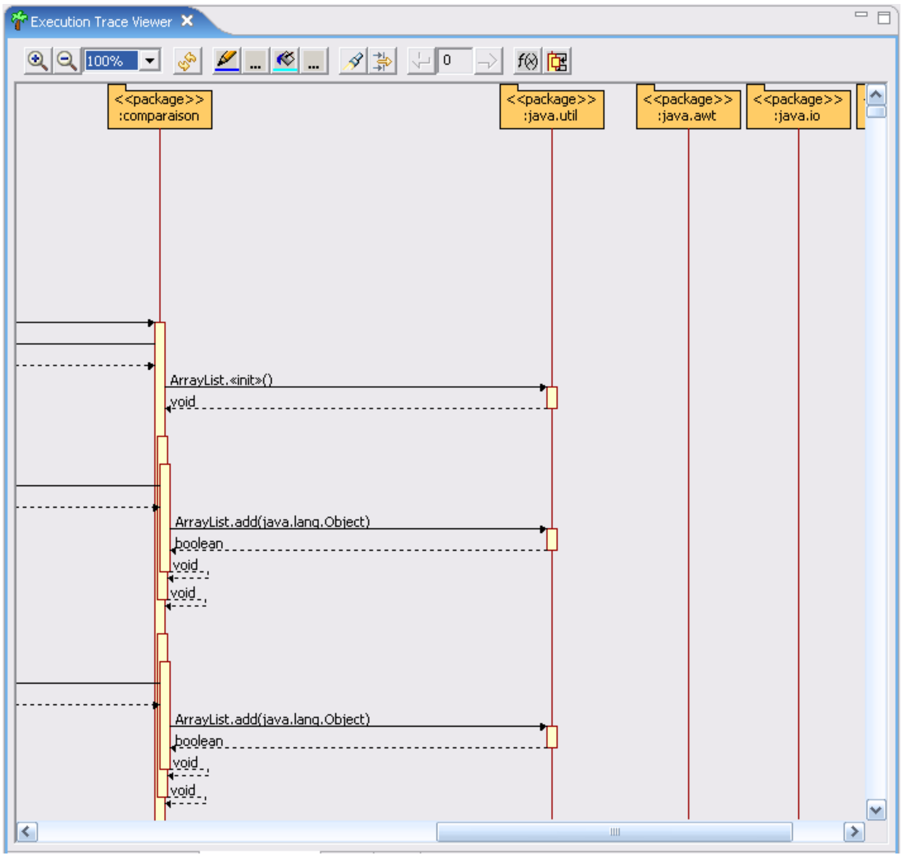


Figure 21. *Sequence Diagram without Filtering Applied*

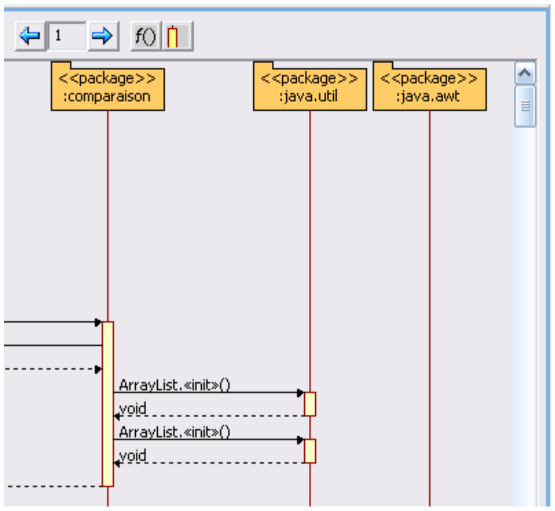


Figure 22. *Sequence Diagram with Filtering Applied*

5.4 Domain Knowledge Definition and Exploitation

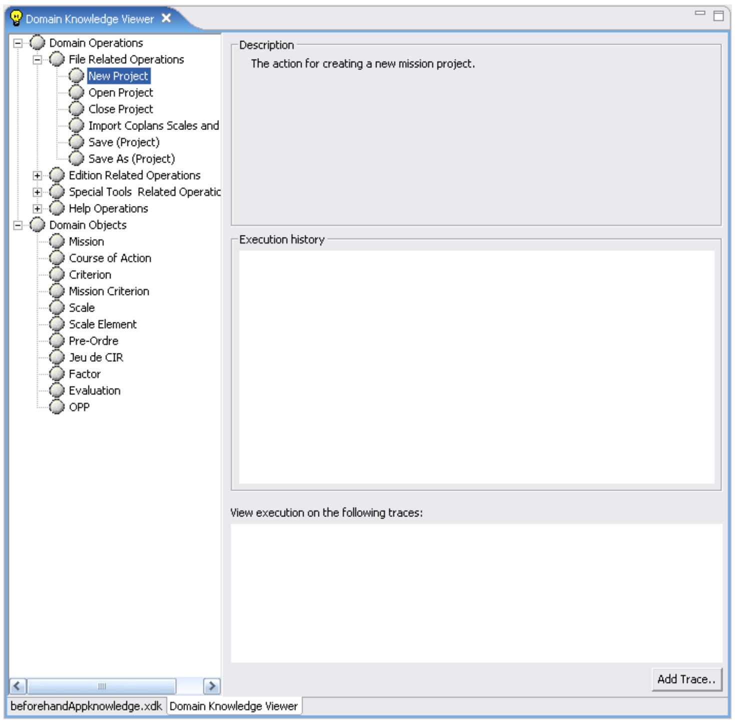
One strategy used by programmers to understand the source code of an unfamiliar system is the top-down approach [35]. Using this strategy, the knowledge about the application domain is first reconstructed and then mapped on the source code. This approach is required to reconstruct and understand a software system at the architectural level, as it allows to map source code elements to their corresponding operational concepts.

One limitation of the current tools which was identified in the state-of-the-art survey previously conducted on architecture recovery and comprehension [5] was that they do not offer functionalities to incorporate domain and user knowledge. This might therefore prevent programmers from using the top-down approach as a comprehension strategy. To overcome this limitation, OASIS v1 offers a way to define and store the domain model of a software system to comprehend. This results in a vocabulary of terms representing entities of the domain and their relationships, which together imply certain semantic information. These entities can then be mapped to an execution trace to determine which ones were executed by a particular run of the system.

Figure 23 on the next page shows the Domain Knowledge Viewer in OASIS v1. On the left-hand side, the operations and objects of the application domain are listed in a tree structure. In the present case, the application under study is a module of the Collaborative Operations Planning System (COPlanS) [36]. COPlanS is an integrated flexible suite of planning, decision-aid, and workflow management tools aimed at supporting the CF Operational Planning Process (OPP). It was developed by the Decision Support Systems Section at DRDC Valcartier. In the present case, examples of domain operations and objects could be respectively "Add a Course of Action (COA)" and "Mission." For each domain operation and object, a description is provided on top of the right pane.

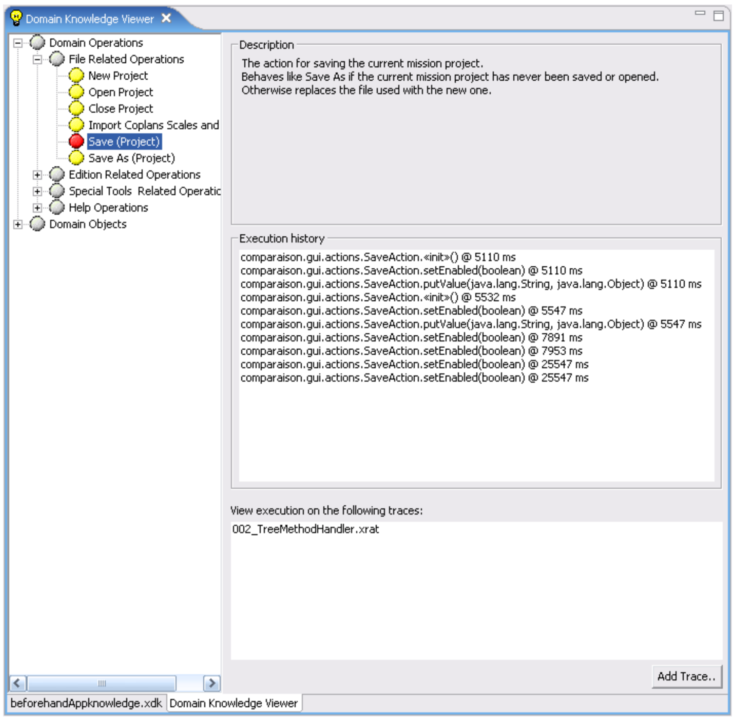
As illustrated in Figure 24, by clicking on the `Add Trace…` button located on the lower right of the viewer, a user can select an execution trace file, in this case, `002_TreeMethodHandler.xrat`, and view the execution history of the methods associated with a domain operation or object. In the present example, the methods executed by the `Save(Project)` operation are displayed. For each method, its fully qualified name and the time, expressed in milliseconds, at which the method was called are indicated.

Once an execution trace file has been selected, the domain operations and objects' nodes become color-coded to indicated whether they were slightly (green), moderately (yellow), or strongly (red) solicited by a particular execution. The color of a node depends whether is was executed by one to three (green), four to five (yellow), or more than six (red) method calls. A node in gray means that it was not executed by any method.



The action for creating a new mission project.

Figure 23. *Domain Knowledge Viewer*



The action for saving the current mission project.
Behaves like Save As if the current mission project has never been saved or opened.
Otherwise replaces the file used with the new one.

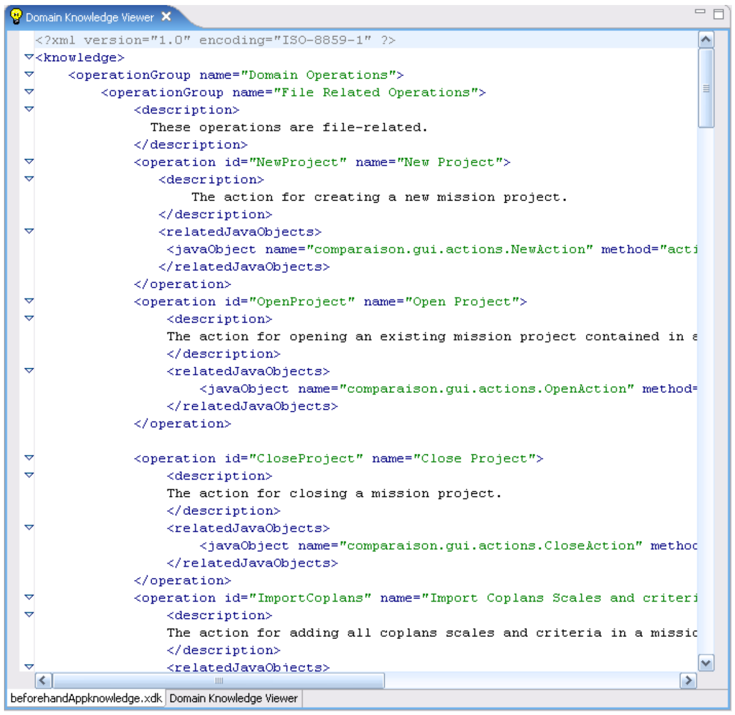
```
comparaison.gui.actions.SaveAction.<init>() @ 5110 ms
comparaison.gui.actions.SaveAction.setEnabled(boolean) @ 5110 ms
comparaison.gui.actions.SaveAction.putValue(java.lang.String, java.lang.Object) @ 5110 ms
comparaison.gui.actions.SaveAction.<init>() @ 5532 ms
comparaison.gui.actions.SaveAction.setEnabled(boolean) @ 5547 ms
comparaison.gui.actions.SaveAction.putValue(java.lang.String, java.lang.Object) @ 5547 ms
comparaison.gui.actions.SaveAction.setEnabled(boolean) @ 7891 ms
comparaison.gui.actions.SaveAction.setEnabled(boolean) @ 7953 ms
comparaison.gui.actions.SaveAction.setEnabled(boolean) @ 25547 ms
comparaison.gui.actions.SaveAction.setEnabled(boolean) @ 25547 ms
```

002_TreeMethodHandler.xrat

Figure 24. *Domain Knowledge Viewer for a Particular Execution Trace*

In OASIS v1, the definition of an application domain is done manually using an eXtensible Markup Language (XML) file. Figure 25 below shows an example of such a file. The association of a domain operation or object to a method, which later allows its mapping to method calls in the execution trace, is also done manually in the XML file.

Despite the above limitations, it was believed important to partially implement this functionality in OASIS v1. It would allow demonstrating to potential military clients the usefulness of being able to associate the high level operational concepts of a software system's application domain to their corresponding entities in execution traces.



```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<knowledge>
    <operationGroup name="Domain Operations">
        <operationGroup name="File Related Operations">
            <description>
              These operations are file-related.
            </description>
            <operation id="NewProject" name="New Project">
               <description>
                   The action for creating a new mission project.
               </description>
               <relatedJavaObjects>
                <javaObject name="comparaison.gui.actions.NewAction" method="acti
               </relatedJavaObjects>
            </operation>
            <operation id="OpenProject" name="Open Project">
                <description>
                <The action for opening an existing mission project contained in a
                </description>
                <relatedJavaObjects>
                    <javaObject name="comparaison.gui.actions.OpenAction" method=
                </relatedJavaObjects>
            </operation>

            <operation id="CloseProject" name="Close Project">
                <description>
                The action for closing a mission project.
                </description>
                <relatedJavaObjects>
                    <javaObject name="comparaison.gui.actions.CloseAction" methoc
                </relatedJavaObjects>
            </operation>
            <operation id="ImportCoplans" name="Import Coplans Scales and criteri
                <description>
                The action for adding all coplans scales and criteria in a missic
                </description>
                <relatedJavaObjects>
```

Figure 25. *Domain Knowledge XML File*

6. OASIS v1 Implementation

The software development methodology selected for OASIS v1 was rapid prototyping. Rapid prototyping is a method for addressing problems in the design and development of systems via prototypes [37]. A prototype is a simplified model of a proposed system. The reason it was decided to develop a prototype for the first implementation of OASIS is because it would help in [38]:

1. Formulating and evaluating requirements, specifications, and designs.

2. Demonstrating the feasibility, behavior, and performance of the proposed system.

3. Identifying and reducing risks of system misdevelopment.

4. Communicating ideas, especially among diverse groups.

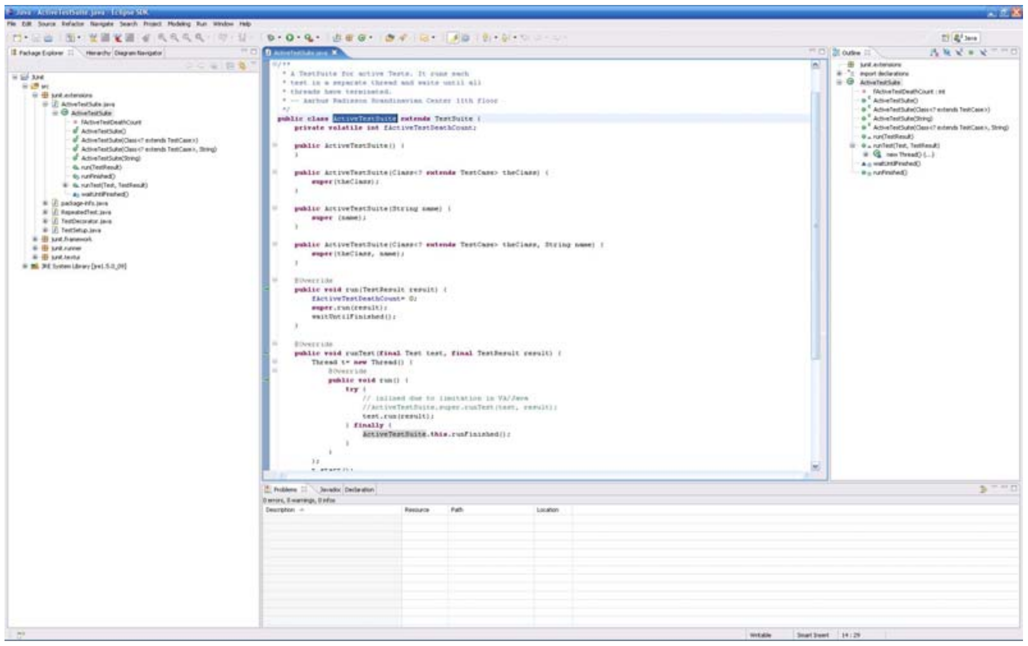
5. Answering questions about specific properties of the proposed system.

In addition to using rapid prototyping as the software development methodology, OASIS v1 also reused as much as possible other components, such as existing third-party libraries and Eclipse plug-ins. The present section introduces the most important components which were reused for the development of OASIS v1.

6.1 Eclipse

Eclipse “is an open source community whose projects are focused on providing a vendor-neutral open development platform and application frameworks for building software” [39]. It is led by the Eclipse Foundation, “a not-for-profit corporation formed to advance the creation, evolution, promotion, and support of the Eclipse Platform and to cultivate both an open source community and an ecosystem of complementary products, capabilities, and services” [39].

The principal advantage of using Eclipse for the development of the first version of the OASIS prototype is that it provides a plug-in based framework that makes it easier to create, integrate, and utilize software tools. A plug-in provides functionality by hooking into extension points defined by other plug-ins. It can also define new extension points. The Eclipse runtime component defines the plug-in infrastructure to discover the available plug-ins at start-up and manage the plug-in loading. By collaborating and exploiting the core Eclipse integration technology, the OASIS project could leverage the numerous plug-ins that composed the Eclipse platform, as well as the plethora of additional plug-ins developed by the Eclipse community. It could therefore concentrate on core competencies to create new development technology, such as the reverse engineering of UML sequence diagrams from Java software systems. Figure 26 on the next page shows a screenshot of the Eclipse workbench which defines the Eclipse user interface paradigm. The version of Eclipse used for OASIS v1 was Eclipse 3.1 milestone 2.



```
/**
 * A TestSuite for active Tests. It runs each
 * test in a separate thread and waits until all
 * threads have terminated.
 * -- Aarhus Radisson Scandinavian Center 11th floor
 */
public class ActiveTestSuite extends TestSuite {
    private volatile int fActiveTestDeathCount;

    public ActiveTestSuite() {
    }

    public ActiveTestSuite(Class<? extends TestCase> theClass) {
        super(theClass);
    }

    public ActiveTestSuite(String name) {
        super (name);
    }

    public ActiveTestSuite(Class<? extends TestCase> theClass, String name) {
        super(theClass, name);
    }

    @Override
    public void run(TestResult result) {
        fActiveTestDeathCount= 0;
        super.run(result);
        waitUntilFinished();
    }

    @Override
    public void runTest(final Test test, final TestResult result) {
        Thread t= new Thread() {
            @Override
            public void run() {
                try {
                    // inlined due to limitation in VA/Java
                    //ActiveTestSuite.super.runTest(test, result);
                    test.run(result);
                } finally {
                    ActiveTestSuite.this.runFinished();
                }
            }
        };
        t.start();
```

Figure 26. *Eclipse User Workbench*

6.2 Infrastructure and Technologies

Figure 27 below illustrates the infrastructure of OASIS v1 and lists the existing Eclipse plug-ins and external libraries which were reused for its implementation.

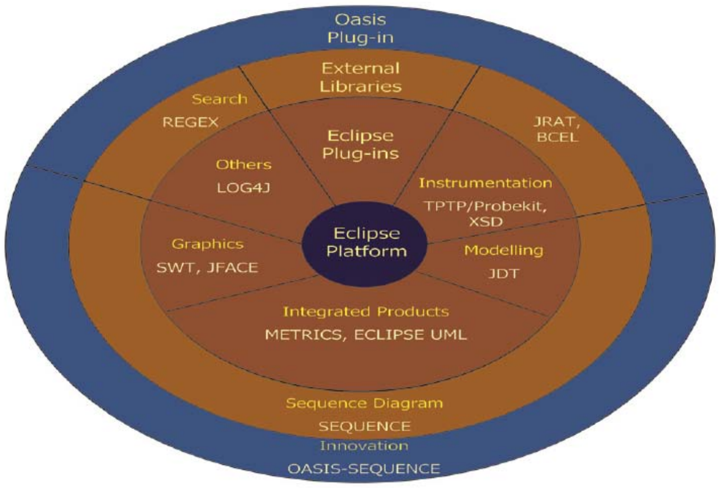


Figure 27. *OASIS v1 Infrastructure and Technologies*

At the center of the previous diagram, there is the Eclipse platform on which OASIS v1 is based. Everything located in the inner brown circle corresponds to existing Eclipse plug-ins used by the prototype. Their functionality is indicated in yellow (e.g., instrumentation, modeling, etc.). The part in lighter brown corresponds to third-party libraries that are also required by the prototype. The rest of this section explains the purpose of each of these components.

6.3 Bytecode Instrumentation

As mentioned in Section 5.1, the selected approach for the collection of dynamic information was Java bytecode instrumentation. Two options were considered for its implementation: the Java Runtime Analysis Toolkit (JRat) [40] and the Eclipse TPTP project. Each of these options is discussed next.

6.3.1 JRat

JRat was implemented to enable developers to better understand the runtime behavior of their Java programs [40]. It can accumulate timing statistics, create trace logging, and track the rate methods are called, as well as their response time. Although there is a number of ways JRat can monitor applications, the one used for OASIS v1 was bytecode instrumentation. JRat uses the Byte Code Engineering Library (BCEL) [41] to manipulate Java bytecode. Figure 29 on the next page shows an example [40] of the code added by JRat for the instrumentation of the method displayed in Figure 28 below.

```
public class MyMath {

    public int max(int a, int b) {
        return (a > b) ? a : b;
    }
}
```

Figure 28. *Java Source Code*

```
public class MyMath {

  static final MethodHandler HANDLER_FOR_max_0 =
    HandlerFactory.getMethodHandler(
                         "org.package.MyMath",
                         "max", "(II)I");

  private final int max__shiftone_JRat(int a, int b) {
    return (a > b) ? a : b;
  }

  public int max(int a, int b) {

    long    start   = 0L;
    Object  args[]  = null;
    boolean success = false;
    try {

      HANDLER_FOR_max_0.onMethodStart(this, args);
       start      = System.currentTimeMillis();
       int result = max__shiftone_JRat(a,b);
       success    = true;
       return result;

    } catch(Throwable t) {

      HANDLER_FOR_max_0.onMethodError(this, args, t);
       throw t;

    } finally {

       HANDLER_FOR_max_0.onMethodFinish(this,
         args, null,
         System.currentTimeMillis() - start,
         success);
    }
  }
}
```

Figure 29. *Java Source Code Instrumented Using JRat*

Given a class file to instrument, JRat will add instrumentation code to all methods except the constructor [42]. However, in order to generate valid sequence diagrams, the class constructor should also be instrumented, as it might call other methods. If not, the hierarchy of method calls will not be captured correctly. Because of this limitation, another option was selected for bytecode instrumentation.

6.3.2 The Eclipse TPTP Project

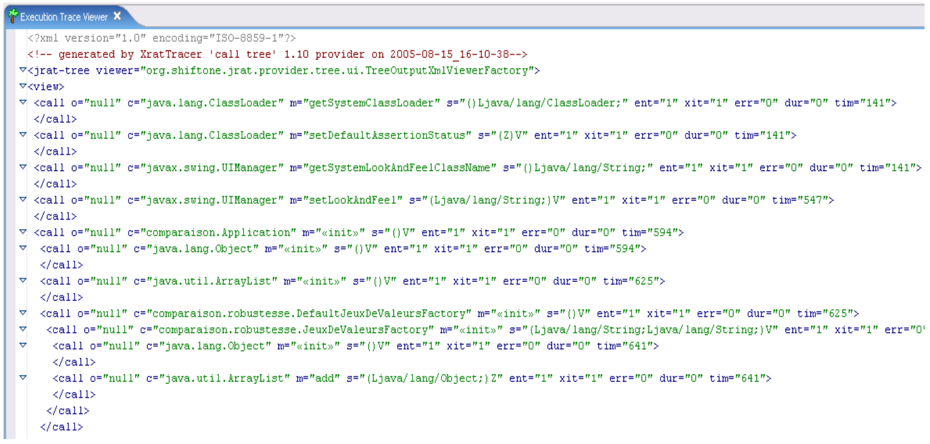
The Eclipse TPTP project provides Probekit, a framework to write and use probes. Probes are Java code fragments that can be invoked at specified points in a Java class file to collect runtime data about an application. Probekit offers various injection points for probes such as method entry, method exit, catch-finally blocks, and class loading. Although Probekit can be used for profiling and debugging, it was used in OASIS v1 to trace method invocations based on certain actions for the purpose of reverse engineering.

A probe is composed of a target, an import, and one or more fragments. The target specifies the filtering criteria for the probe, i.e., the packages, classes, and methods that should be targeted by the probe. The import is used to specify the Java packages and classes that are referenced by the probe. Finally, the fragment defines the probe logic, i.e., where the fragment code will get injected into the application and the Java code that should be invoked by the fragment.

Once a probe has been created, it can be used to instrument Java applications statically. With static instrumentation, probes are injected into the selected class files prior to execution. The Java classes can then be executed normally and have the probes collect the necessary runtime data.

6.3.3 XRat File Format

The execution of Java bytecode instrumented using probes generates a tree of method calls, capturing their order and context, stored as a XRat file. XRat is the format used by JRat to log execution traces as XML documents. This format was selected since Probekit does not provide any particular file format to store execution traces. Figure 30 below shows a section of a XRat file.



```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- generated by XratTracer 'call tree' 1.10 provider on 2005-08-15_16-10-38-->
<jrat-tree viewer="org.shiftone.jrat.provider.tree.ui.TreeOutputXmlViewerFactory">
<view>
 <call o="null" c="java.lang.ClassLoader" m="getSystemClassLoader" s="()Ljava/lang/ClassLoader;" ent="1" xit="1" err="0" dur="0" tim="141">
 </call>
 <call o="null" c="java.lang.ClassLoader" m="setDefaultAssertionStatus" s="(Z)V" ent="1" xit="1" err="0" dur="0" tim="141">
 </call>
 <call o="null" c="javax.swing.UIManager" m="getSystemLookAndFeelClassName" s="()Ljava/lang/String;" ent="1" xit="1" err="0" dur="0" tim="141">
 </call>
 <call o="null" c="javax.swing.UIManager" m="setLookAndFeel" s="(Ljava/lang/String;)V" ent="1" xit="1" err="0" dur="0" tim="547">
 </call>
 <call o="null" c="comparaison.Application" m="«init»" s="()V" ent="1" xit="1" err="0" dur="0" tim="594">
  <call o="null" c="java.lang.Object" m="«init»" s="()V" ent="1" xit="1" err="0" dur="0" tim="594">
  </call>
  <call o="null" c="java.util.ArrayList" m="«init»" s="()V" ent="1" xit="1" err="0" dur="0" tim="625">
  </call>
  <call o="null" c="comparaison.robustesse.DefaultJeuxDeValeursFactory" m="«init»" s="()V" ent="1" xit="1" err="0" dur="0" tim="625">
   <call o="null" c="comparaison.robustesse.JeuxDeValeursFactory" m="«init»" s="(Ljava/lang/String;Ljava/lang/String;)V" ent="1" xit="1" err="0
    <call o="null" c="java.lang.Object" m="«init»" s="()V" ent="1" xit="1" err="0" dur="0" tim="641">
    </call>
    <call o="null" c="java.util.ArrayList" m="add" s="(Ljava/lang/Object;)Z" ent="1" xit="1" err="0" dur="0" tim="641">
    </call>
   </call>
  </call>
```

Figure 30. *Sample XRat File*

For each method call, represented by a `<call>` tag, the following attributes are stored:

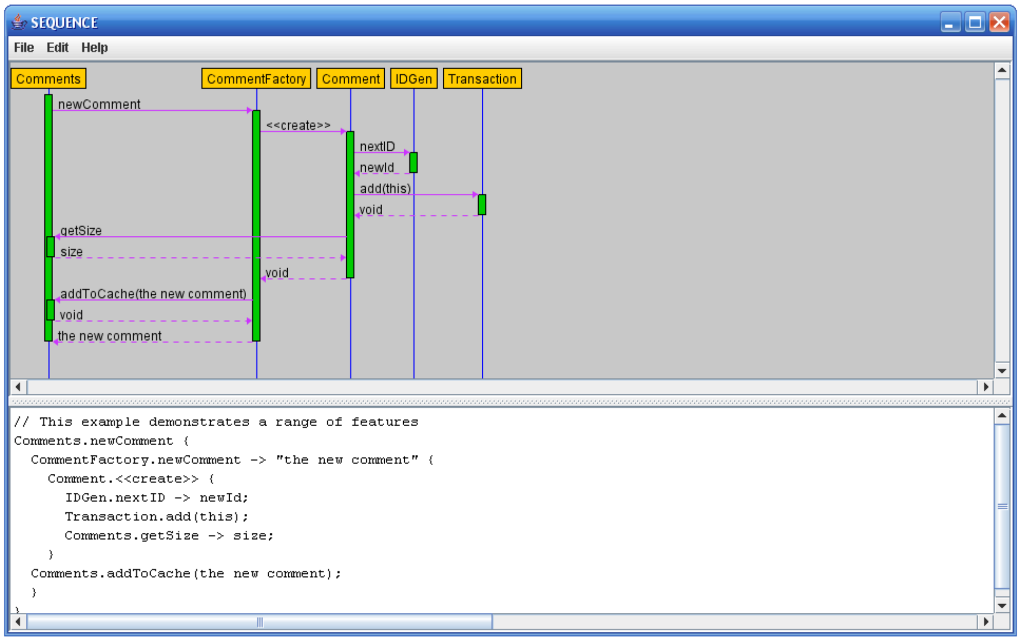
Table 2. *XRat `<call>` Tag Attributes*

ATTRIBUTE	DESCRIPTION
c	The fully qualified name of the class to which the method belongs.
m	The method name.
s	The method signature expressed using Java Native Interface (JNI) descriptors.
tim	The time, expressed in milliseconds, at which the method was called.

Although the `<call>` tag has also the `o`, `ent`, `xit`, `err` and `dur` attributes, those are not used. The reason why default values are put for them is because they are required by the Document Type Definition (DTD) of the XRat document.

6.4 SEQUENCE

SEQUENCE [43] is an open source Java library for generating UML sequence diagrams. Unlike similar programs used in forward engineering, it does not require the user to actually draw the diagram. Instead, the user writes a textual description of the method calls to model following the SEQUENCE syntax. Then, the layout and diagram are respectively calculated and drawn automatically. For instance, the sequence diagram on the next page was defined by the description beneath it. This example demonstrates most of the language’s features available to describe diagrams using the SEQUENCE syntax.



```
// This example demonstrates a range of features
Comments.newComment {
  CommentFactory.newComment -> "the new comment" {
    Comment.<<create>> {
      IDGen.nextID -> newId;
      Transaction.add(this);
      Comments.getSize -> size;
    }
  Comments.addToCache(the new comment);
  }
}
```

Figure 31. *SEQUENCE*

6.5 Integrating SEQUENCE into Eclipse

When SEQUENCE was integrated into Eclipse as a plug-in, one technical challenge was faced. This was due to the fact that SEQUENCE and Eclipse are based on two different widget toolkits, which are respectively AWT/Swing and SWT.

The Standard Widget Toolkit (SWT) is “the software component that delivers native widget functionality for the Eclipse platform in an operating independent manner” [44]. Although SWT is similar to AWT/Swing, they differ in their implementation strategy. “SWT uses native widgets wherever possible for three main advantages: performance, look-and-feel, and debugging” [44]. It only implements a widget in Java if there is no native version available. While SWT offers the same Application Programming Interface (API) on all platforms, its implementation differs for each of them. This implementation is partially contained in a shared library which offers a subset of the operating system widget API to SWT using JNI [44]. On Windows, this library is implemented as a DLL.

The Abstract Window Toolkit (AWT) follows a different approach than SWT. It adopts a least common denominator strategy by providing only the widgets which are available on all platforms [44]. Swing provides the missing widgets by implementing in Java higher-level ones on top of the existing AWT widgets. As a result, contrary to SWT, Swing provides only one implementation for all platforms [44].

At the time OASIS v1 was implemented, there was an experimental mechanism to run Swing-based tools within Eclipse. However, its use was not officially supported and it had the following limitations [45]:

1. It only worked on Windows.

2. The keyboard did not work for all widgets.

3. There could be possible deadlock problems.

4. The API was internal and subjected to change.

Due to the above restrictions, it was decided to re-implement in SWT the SEQUENCE classes which used the AWT/Swing widgets to graphically render the sequence diagrams. This solution also provided the tightest interface integration mechanism [44]: SEQUENCE would run in-place within Eclipse, i.e., it would be visually indistinguishable from the built in tools. Also, it would run in-process, i.e., within the same virtual machine and class libraries as Eclipse. Furthermore, it could achieve seamless functional integration in the case the Eclipse API is used.

In addition to the above partial re-implementation, SEQUENCE was also modified to support the functionalities described in Section 5.3, i.e., package level sequence diagrams, method collapsing, searching, and filtering.

6.6 Additional Libraries and Plug-ins

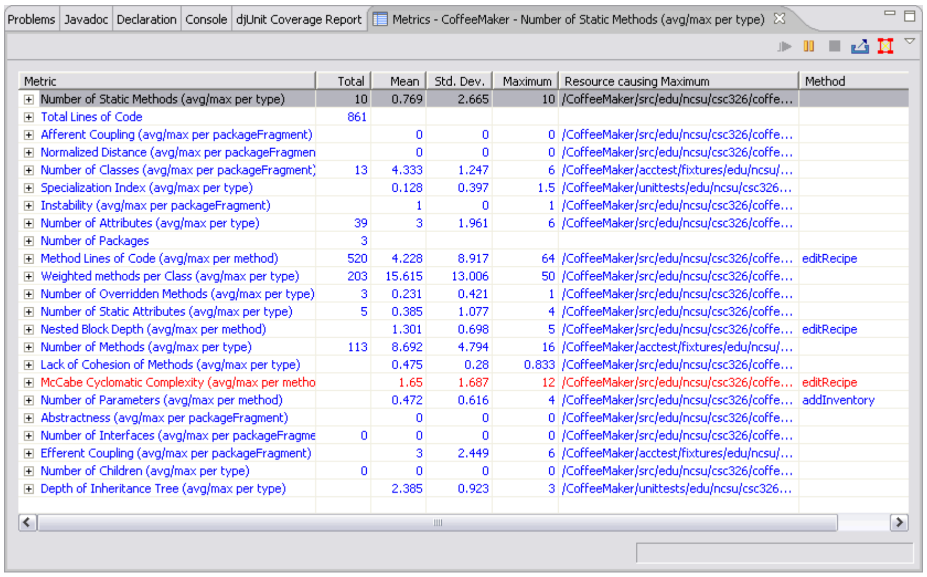
As already mentioned, one advantage of using the Eclipse platform for the first version of the OASIS prototype was that its numerous plug-ins as well as the others developed by the Eclipse community could be leveraged. This section presents the existing plug-ins and other libraries which were reused for the implementation of OASIS v1.

6.6.1 Metrics

A metric measures a property of a piece of software or its specifications. It has been shown that metrics can provide guidance in analyzing the quality of the design and source code of a system, as well as its possible maintainability and comprehension [46]. For this reason, it was decided to integrate within OASIS v1 the open source Eclipse plug-in Metrics [47].

As its name indicates, this plug-in provides metrics calculation and dependency analysis for the Eclipse platform. It measures various metrics with average and standard deviation, as well as detects cycles in package and type dependencies and graphs them. Figure 32 on the next page displays one of the Eclipse views provided by the Metrics plug-in.

The Metrics view displayed in Figure 32 provides 23 metrics. For a complete list, refer to [48]. Metrics which are out of optimal range are highlighted in red. Otherwise, they are displayed in blue, if they are within the range, and in black, if their value cannot be calculated.


Problems
Javadoc
Declaration
Console
djUnit Coverage Report
Metrics - CoffeeMaker - Number of Static Methods (avg/max per type)

Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum	Method
Number of Static Methods (avg/max per type)	10	0.769	2.665	10	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	
Total Lines of Code	861					
Afferent Coupling (avg/max per packageFragment)		0	0	0	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	
Normalized Distance (avg/max per packageFragmen		0	0	0	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	
Number of Classes (avg/max per packageFragment)	13	4.333	1.247	6	/CoffeeMaker/acctest/fixtures/edu/ncsu/...	
Specialization Index (avg/max per type)		0.128	0.397	1.5	/CoffeeMaker/unittests/edu/ncsu/csc326...	
Instability (avg/max per packageFragment)		1	0	1	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	
Number of Attributes (avg/max per type)	39	3	1.961	6	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	
Number of Packages	3					
Method Lines of Code (avg/max per method)	520	4.228	8.917	64	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	editRecipe
Weighted methods per Class (avg/max per type)	203	15.615	13.006	50	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	
Number of Overridden Methods (avg/max per type)	3	0.231	0.421	1	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	
Number of Static Attributes (avg/max per type)	5	0.385	1.077	4	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	
Nested Block Depth (avg/max per method)		1.301	0.698	5	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	editRecipe
Number of Methods (avg/max per type)	113	8.692	4.794	16	/CoffeeMaker/acctest/fixtures/edu/ncsu/...	
Lack of Cohesion of Methods (avg/max per type)		0.475	0.28	0.833	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	
McCabe Cyclomatic Complexity (avg/max per metho		1.65	1.687	12	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	editRecipe
Number of Parameters (avg/max per method)		0.472	0.616	4	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	addInventory
Abstractness (avg/max per packageFragment)		0	0	0	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	
Number of Interfaces (avg/max per packageFragme	0	0	0	0	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	
Efferent Coupling (avg/max per packageFragment)		3	2.449	6	/CoffeeMaker/acctest/fixtures/edu/ncsu/...	
Number of Children (avg/max per type)	0	0	0	0	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	
Depth of Inheritance Tree (avg/max per type)		2.385	0.923	3	/CoffeeMaker/unittests/edu/ncsu/csc326...	

Figure 32. *Eclipse View Provided by the Metrics Plug-in*

Although Metrics provides standard source and object-oriented class metrics, they are not necessarily architecturally significant. The following list of object-oriented package metrics, also provided by the Metrics plug-in, has been proved in [49] to be particularly useful for architecture recovery and comprehension. This suite of metrics is based on the work undertaken by Martin [50].

Afferent Coupling (Ca). Counts the number of other packages which depend on classes within the analyzed package. Ca is an indicator of the level of responsibility of a package.

Efferent Coupling (Ce). Counts the number of other packages that the classes within the analyzed package depend upon. Ce is an indicator of the package's independence.

Abstractness (A). It is the ratio of the number of abstract classes within a package relative to the total number of classes it contains. The range of this metric is from 0 to 1. An abstractness value of zero (A = 0) indicates a completely concrete package, while a value of one (A = 1) indicates a completely abstract package.

Instability (I). Instability is defined as the ratio between efferent and total coupling (Ca + Ce). This metric is an indicator of the package's resilience to change, i.e., the effort to change a package without impacting other packages within the application. The range of this metric goes from 0 to 1. An I of 0 reveals a completely stable package, while an I of 1 indicates that the package is instable.

Distance from the Main Sequence (DMS). Calculates the perpendicular distance of a package from the idealized line given by A + I = 1. It indicates the package's balance between abstractness and stability. A package squarely on the main sequence is perfectly balanced with respect to abstractness and stability. Ideally, packages should either be completely abstract and stable (x = 0, y = 1), or completely concrete and instable (x = 1, y = 0). The range for this metric goes from 0 to 1. A DMS of 0 indicates that a package is coincident with the main sequence, while a DMS of 1 reveals that the package is as far as possible from the main sequence.

6.6.2 Omondo EclipseUML

As mentioned in Section 4.1.1, Omondo EclipseUML is an Eclipse plug-in for model driven development based on UML and EMF. In addition to its capacity to reverse engineer sequence diagrams previously described, it also offers functionalities to reverse engineer class and package diagrams.

The reason why EclipseUML was added to OASIS v1 is because the behavioral views provided by the latter could be complemented by the structural ones of EclipseUML. At the architectural level, the structural views which are of most interest are package and class diagrams. Figure 33 on the next page displays the package diagram of JUnit reverse engineered using EclipseUML.

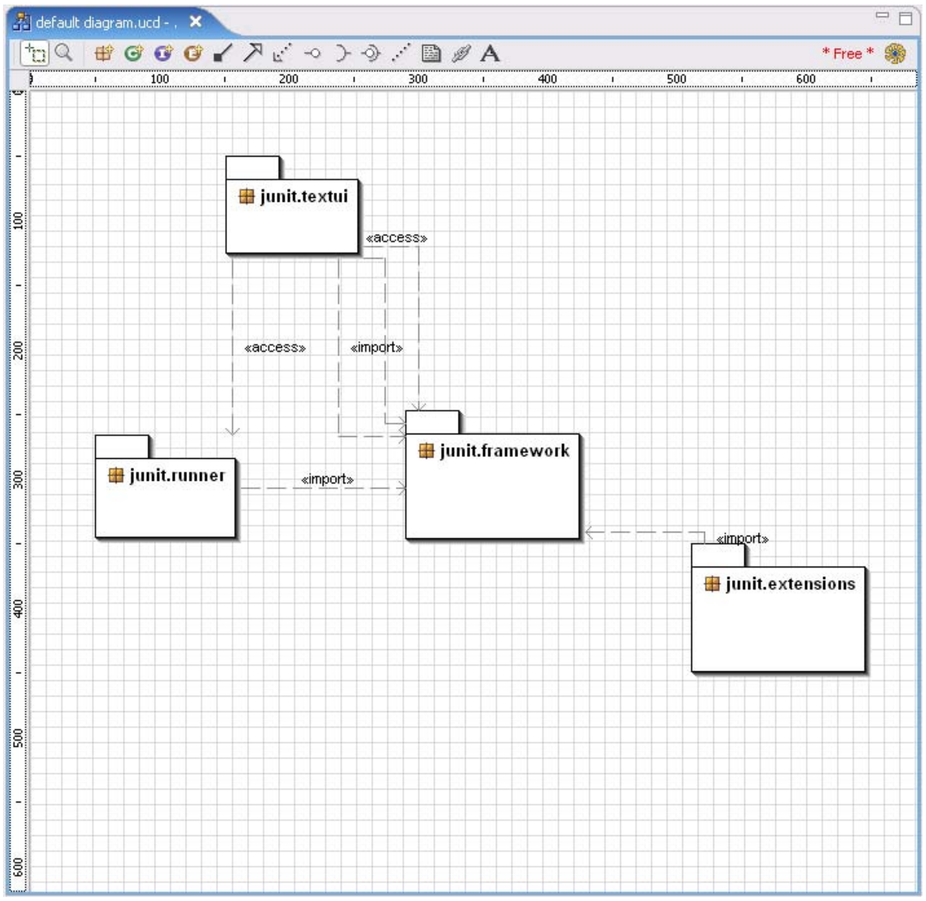


Figure 33. *Package Diagram Reverse Engineered by EclipseUML*

6.6.3 The Eclipse JDT Subproject

The Eclipse Java Development Tools (JDT) subproject [51] "contributes a set of plug-ins that adds the capabilities of a full-featured Java IDE to the Eclipse platform. The JDT plug-ins provide APIs so that they can themselves be further extended by other tool builders" [52]. The JDT API was used extensively for the development of the different Eclipse plug-ins which implement the functionalities of OASIS v1.

6.6.4 XSD

The XML Schema Definition (XSD) library is part of the Eclipse Model Development Tools (MDT) project [53]. It provides an API for manipulating the components of an XML Schema, as described by the World Wide Web Consortium (W3C) XML Schema specifications [54], as well as for manipulating the DOM-accessible representation of XML and keeping these representations in agreement as schemas are modified [55]. The XSD library was used in OASIS v1 as it is required by TPTP.

6.6.5 Log4j

Log4j [56] is an open source Java-based library developed by the Apache Software Foundation [57]. It allows to insert log statements into an application. With Log4j, logging can be enabled at runtime without modifying the application bytecode. It was designed so that even though log statements remain in the bytecode, they do not heavily impact the performance of the application. The Log4j library was used in OASIS v1 as it is required by SEQUENCE.

7. Conclusions and Future Work

Reverse engineering techniques and tools are required to understand, at the architectural level, the structure and behavior of software systems for which the documentation is neither existent nor up to date. The proof of concept prototype presented in this document is the first step towards the complete reverse engineering of software systems behavior at the architectural level. The reverse engineering of UML sequence diagrams can play an important part in the maintenance of CF object-oriented software systems. These systems are considerable investments which will have to be maintained over a long period of time, without the assistance of the original designers and developers and most of the time, with incomplete documentation. Reverse-engineered sequence diagrams provide essential insights for the comprehension and maintenance of such systems, as object interactions are at the core of object-oriented design and programming [58].

The advantage of the proposed approach over the already existing solutions is that the sequence diagrams can be reconstructed at the package level, in addition to the object level. Sequence diagrams displayed at the package level allow mapping system functionalities to source code and architectural entities. This is essential to understand unfamiliar large scale systems at the architectural level. Also, with the prototype being integrated into Eclipse, the system behavioral views it reverse engineers can be complemented by structure views provided by other tools.

Following the implementation of this proof of concept prototype, another one, with an improved set of functionalities as well as additional views, should be developed. Also, a suitable meta-model should be used to represent and store the execution data of Java software systems, instead of the .xrat file format. Ideally, once this second prototype is implemented, another study, similar to the one previously conducted as part of the OASIS project [6], but with an improved design and set of comprehension tasks, should be conducted. Its objective would be to assess the added value of the OASIS architecture recovery and comprehension prototype on the understanding of participants. Future work should also consist of extending the prototype to support the architecture recovery and comprehension of C/C++ legacy systems, through the Eclipse C/C++ Development Tooling (CDT) project [59]. This would address one limitation of most existing tools, i.e., multi-language support.

8. References

1. The Technical Cooperation Program - Joint Systems and Analysis Group, “The Engineering and Acquisition of Systems of Systems in the United States DoD,” *Tech. Report TR-JSA-TP4-1-2001*, Jan. 2001.

2. D. Garlan and D.E. Perry, “Introduction to the Special Issue on Software Architecture,” *IEEE Trans. on Software Eng.*, vol. 21 no. 4, Apr. 1995, pp. 269-274.

3. R. Richardson, et al., “A Survey of Research into Legacy System Migration,” *Tech. Report TCD-CS-1997-01*, Trinity College Dublin, Dublin, Ireland, Jan. 1997.

4. M. Lizotte and J. Rilling, “OASIS: Opening-up Architecture of Software-Intensive Systems,” *Proc. of the 24th Army Science Conf. (ASC ’04)*, Orlando, Fla., Nov. 2004.

5. J. Rilling, “State of the Art Report: System Architecture Recovery and Comprehension,” *Tech. Report*, DRDC Valcartier, Val-Bélair, Que., 2003.

6. P. Charland, et al., “Using Software Analysis Tools to Understand Military Applications: A Qualitative Study,” *Tech. Memorandum TM 2005-425*, DRDC Valcartier, Val-Bélair, Que., 2005.

7. P. Charland, et al., “Opening up Architectures of Software-Intensive Systems: A Functional Decomposition,” *Tech. Memorandum TM 2006-732*, DRDC Valcartier, Val-Bélair, Que., 2006.

8. Eclipse, “Eclipse.org home,”Jun. 2007; http://www.eclipse.org/.

9. R. Kazman, L. O’Brien, and C. Verhoef, “Architecture Reconstruction Guidelines, Third Edition,” *Tech. Report CMU/SEI-2002-TR-034*, Carnegie Mellon Univ., Pittsburgh, Pa., Nov. 2003.

10. Rational PurifyPlus, “IBM - Rational PurifyPlus - Rational PurifyPlus - Software,” Jun. 2007; http://www-306.ibm.com/software/awdtools/purifyplus/win/.

11. CASE ATTI, “CASE ATTI: A Testbed for Sensor Data Fusion,” Jun. 2007; http://www.valcartier.drdc-rddc.gc.ca/poolpdf/e/137_e.pdf.

12. B. Bruegge and A.H. Dutoit, *Object-Oriented Software Engineering: Conquering Complex and Changing Systems*, Prentice Hall, 2000.

13. G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 2005.

14. Visual Paradigm, “UML 2 Diagrams - UML Modeling Tool,” Jun. 2007; http://www.visual-paradigm.com/VPGallery/diagrams/.

15. OMG, “OMG Unified Modeling Language Specification Version 1.5,” Object Management Group, Mar. 2003.

16. E. Korshunova, et al., “CPP2XMI: Reverse Engineering of UML Class, Sequence, and Activity Diagrams from C++ Source Code,” *Proc. of the 13th Working Conf. on Reverse Eng. (WCRE 2006)*, Benevento, Italy, Oct. 2006, pp. 297-298.

17. EclipseUML, “Omondo Corp - The Modeling Eclipse UML Model Driven Tool,” Jun. 2007; http://www.omondo.com/.

18. Sequence Diagram Viewer - NetBeans Module, “sequencediagramviewer: Sequence Diagram Viewer - NetBeans Module,” Jun. 2007; https:// sequencediagramviewer.dev.java.net/.

19. Borland Together, “Software Architecture Design, Visual UML & Business Process Modeling - from Borland,” Jun. 2007; http://www.borland.com/ us/products/ together.

20. NetBeans UML Modeling, “uml: netbeans.org : UML Modeling,” Jun. 2007; http://uml.netbeans.org/index.html.

21. JUnit, “JUnit, Testing Resources for Extreme Programming,” Jun. 2007; http://www.junit.org/.

22. NetBeans IDE, “Welcome to NetBeans,” Jun. 2007; http://www.netbeans.org/.

23. L.C. Briand, Y. Labiche, and J. Leduc, “Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software,” *IEEE Trans. on Software Eng.*, vol. 32, no. 9, Sept. 2006, pp. 642-663.

24. C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*, Prentice Hall, 2003.

25. M. Merdes and D. Dorsch, “Experiences with the Development of a Reverse Engineering Tool for UML Sequence Diagrams: A Case Study in Modern Java Development,” *Proc. of the 4th Int’l Symposium on Principles and Practice of Programming in Java (PPPJ ’06)*, Manheim, Germany, Aug. 2006, pp. 125-134.

26. J2U, “Nasra.fr,” Jun. 2007; http://www.nasra.fr/j2u.html.

27. Eclipse TPTP, “Eclipse Test & Performance Tools Platform Project,” Jun. 2007; http://www.eclipse.org/tptp/.

28. *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Press, 1990.

29. A. Vasconcelos, R. Cepêda, And C. Werner, “An Approach to Program Comprehension through Reverse Engineering of Complementary Software Views,” *1st Int’l Workshop on Program Comprehension through Dynamic Analysis (PCODA)*, Pittsburgh, Pa., Nov. 2005, pp. 58-62.

30. A. Hamou-Lhadj and T.C. Lethbridge, “A Survey of Trace Exploration Tools and Techniques,” *Proc. of the 2004 Conf. of the Centre for Advanced Studies on Collaborative Research*, Markham, Ont. Oct. 2004, pp. 42-55.

31. W. Löwe, A. Ludwig, and A. Schwind, “Understanding Software - Static and Dynamic Aspects,” *Proc. of the 17th Int’l Conf. on Advanced Science and Technology (ICAST ’01)*, Chicago, Ill., October 2001, pp. 83-88.

32. M. Biberstein, et al., “Instrumenting Annotated Programs,” *Proc. of the 1st ACM/USENIX Int’l Conf. on Virtual Execution Environments*, Chicago, Ill., Jun. 2005, pp. 164-174.

33. J. Guo, Y. Liao, and B. Parviz, “A Performance Validation Tool for J2EE Applications,” *Proc. of the 13th Ann. IEEE Int’l Symp. And Workshop on Eng. of Computer Based Systems*, Potsdam, Germany, Mar. 2006, pp. 387-396.

34. W. Schütz, *The Testability of Distributed Real-Time Systems*, Springer, 1993.

35. R. Brooks, “Towards a Theory of the Comprehension of Computer Programs,” *Int’l J. of Man-Machine Studies*, vol. 18, no. 6, June 1983, pp. 543-554.

36. COPlanS, “COPlanS - Collaborative Operations Planning System,” Jun. 2007; http://www.valcartier.drdc-rddc.gc.ca/poolpdf/e/166_e.pdf.

37. M. Tanik and R. Yeh, “The Role of Rapid Prototyping in Software Development,” *Proc. of the 22nd Hawaii Int’l Conf. on System Sciences*, Kauai, Hawaii, Jan. 1989, pp. 337-338.

38. L. Luqi and R. Steigerwald, “Rapid Software Prototyping,” *Proc. of the 25th Hawaii Int’l Conf. on System Sciences*, Kauai, Hawaii, Jan. 1992, pp. 470-479.

39. Eclipse, “About the Eclipse Foundation,” Jun. 2007; http://www.eclipse.org/org/.

40. JRat, “JRat the Java Runtime Analysis Toolkit,” Jun. 2007; http:// jrat.sourceforge.net/.

41. BCEL, “BCEL - Byte Code Engineering Library (BCEL),” Jun. 2007; http:// jakarta.apache.org/bcel/manual.html.

42. D. Lo and S.-C. Khoo, “SMArTIC: Towards Building an Accurate, Robust and Scalable Specification Miner,” *Proc. of the 14th ACM SIGSOFT Int’l Symp. on Foundations of Software Eng.*, Portland, Ore., Nov. 2006, pp. 265-275.

43. SEQUENCE, “itymbi …: SEQUENCE Archives,” Jun. 2007; http://www.zanthan.com/itymbi/archives/cat_sequence.html.

44. D. Rayside, et al., “Integrating SHriMP with the IBM WebSphere Studio Workbench,” *Proc. of the 9th NRC/IBM Centre for Advanced Studies Conference (CASCON '01)*, Toronto, Ont., Nov. 2001, pp. 79-93.

45. V. Irvine, “Limitations of Swing/SWT Experimental Integration Mechanism,” *Eclipse Corner Newsgroup*, Jul. 2001.

46. N.E. Fenton, “Software Measurement Programs,” *Software Testing and Quality Eng.*, vol. 1, no. 3. 1999, pp. 40-46.

47. Eclipse Metrics plug-in, “SourceForge.net: Eclipse Metrics plugin,” Jun. 2007; http://sourceforge.net/projects/metrics.

48. Eclipse Metrics plug-in, “Metrics 1.3.6,” Jun. 2007; http://metrics.sourceforge.net/.

49. I. Gorton and L. Zhu, “Tool Support for Just-in-Time Architecture Reconstruction and Evaluation: An Experience Report,” *Proc. of the 27th Int’l Conf. on Software Eng. (ICSE '05)*, St. Louis, Mo., May 2005, pp. 514-523.

50. R. Martin, “OO Design Quality Metrics: An Analysis of Dependencies,” *Proc. of the Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics*, Oct. 1994.

51. Eclipse JDT, “Eclipse Java Development Tools (JDT) Subproject,” Jun. 2007; http://www.eclipse.org/jdt/.

52. Eclipse JDT, “Eclipse Java Development Tools (JDT) Subproject Overview,” Jun. 2007; http://www.eclipse.org/jdt/overview.php.

53. Eclipse MDT, “Eclipse Modeling - MDT - Home,” Jun. 2007; http://www.eclipse.org/modeling/mdt/.

54. XML Schema, “XML Schema Part 0: Primer Second Edition,” Jun. 2007; http://www.w3.org/TR/xmlschema-0/.

55. Eclipse MDT, “Eclipse Modeling - MDT - Home,” Jun. 2007; http://www.eclipse.org/modeling/mdt/?project=xsd#xsd.

56. Log4j, “Log4j project - Introduction,” Jun. 2007; http://logging.apache.org/ log4j/.

57. Apache, “Welcome! - The Apache Software Foundation,” Jun. 2007; http://www.apache.org/.

58. A. Rountev, O. Volgin, and M. Reddoch, “Static Control-Flow Analysis for Reverse Engineering of UML Sequence Diagrams,” *Proc. of the 6th ACM*

SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Eng., Lisbon, Portugal, Sept. 2005, pp. 96-102.

59. Eclipse CDT, “Eclipse C/C++ Development Tooling - CDT,” Jun. 2007; http://www.eclipse.org/cdt/.

List of Acronyms

A	Abstractness
API	Application Programming Interface
AWT	Abstract Window Toolkit
BCEL	Byte Code Engineering Library
C4ISR	Command, Control, Communications, Computers, Intelligence, Surveillance, and Reconnaissance
Ca	Afferent Coupling
CASE	Computer-Aided Software Engineering
CASE ATTI	Concept Analysis and Simulation Environment for Automatic Target Tracking and Identification
CDT	C/C++ Development Tooling
Ce	Efferent Coupling
CF	Canadian Forces
COA	Course of Action
COPlanS	Collaborative Operations Planning System
CPF	Canadian Patrol Frigate
DMS	Distance from the Main Sequence
DND	Department of National Defence
DRDC	Defence Research and Development Canada
DTD	Document Type Definition
EDI	Environnement de développement intégré
EMF	Eclipse Modeling Framework
FC	Forces canadiennes

HCI	Human Computer Interface
I	Instability
IDE	Integrated Development Environment
J2U	Java to UML
JDT	Java Development Tools
JNI	Java Native Interface
JRat	Java Runtime Analysis Toolkit
JRE	Java Runtime Environment
MDT	Model Development Tools
OASIS	Opening up Architectures of Software-Intensive Systems
OPP	Operational Planning Process
SoS	System of Systems
SWT	Standard Widget Toolkit
TPTP	Test and Performance Tools Platform
UML	Unified Modeling Language
W3C	World Wide Web Consortium
XML	eXtensible Markup Language
XSD	XML Schema Definition

Distribution List

INTERNAL DISTRIBUTION

1 - Director General

3 - Document Library

1 - Head, System of Systems

1 - Philippe Charland (author)

1 - Dany Dessureault (author)

1 - David Ouellet (author)

1 - Michel Lizotte (author)

1 - Geneviève Dussault

1 - Michel Ducharme

1 - Head, Information and Knowledge Management

1 - François Lemieux

1 - Martin Salois

1 - Head, Decision Support Systems

1 - Bruno Gilbert

1 - Marc Lauzon

1 - Marc Grondin

1 - LCol Pierre Lefebvre

EXTERNAL DISTRIBUTION

1 - DRDKIM (PDF file)

DRDC Headquarters - 305 Rideau Street, Ottawa, ON, K1A 0K2

1 - Director Science and Technology Command, Control, Communications, Computers, Intelligence, Surveillance and Reconnaissance (DSTC4ISR)

1 - Klaus Kollenberg (DSTC4ISR 3)

1 - Donna Wood (DSTC4ISR 4)

1 - Norbert Haché (DSTC4ISR SPO)

1 - Richard Lestage (Director Science and Technology Air 6)

DOCUMENT CONTROL DATA

1. ORIGINATOR (name and address) Defence Research and Development Canada Valcartier 2459, Pie-XI Blvd North Québec, Québec G3J 1X5 Canada	2. SECURITY CLASSIFICATION (Including special warning terms if applicable) Unclassified

3. TITLE (Its classification should be indicated by the appropriate abbreviation (S, C, R or U)
Opening up architectures of software-intensive systems: A first prototype implementation (U)

4. AUTHORS (Last name, first name, middle initial. If military, show rank, e.g. Doe, Maj. John E.)
Charland, Philippe; Dessureault, Dany; Ouellet, David; Lizotte, Michel

5. DATE OF PUBLICATION (month and year) 2007	6a. NO. OF PAGES 62	6b .NO. OF REFERENCES 59

7. DESCRIPTIVE NOTES (the category of the document, e.g. technical report, technical note or memorandum. Give the inclusive dates when a specific reporting period is covered.)
Technical Memorandum

8. SPONSORING ACTIVITY (name and address)
Defence Research and Development Canada Valcartier
2459, Pie-XI Blvd North
Québec, Québec
G3J 1X5 Canada

9a. PROJECT OR GRANT NO. (Please specify whether project or grant) 15ak	9b. CONTRACT NO.
10a. ORIGINATOR'S DOCUMENT NUMBER DRDC Valcartier TM 2006-781	10b. OTHER DOCUMENT NOS N/A

11. DOCUMENT AVAILABILITY (any limitations on further dissemination of the document, other than those imposed by security classification)

- [✓] Unlimited distribution
- [ ] Restricted to contractors in approved countries (specify)
- [ ] Restricted to Canadian contractors (with need-to-know)
- [ ] Restricted to Government (with need-to-know)
- [ ] Restricted to Defense departments
- [ ] Others

12. DOCUMENT ANNOUNCEMENT (any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in 11) is possible, a wider announcement audience may be selected.)

13. ABSTRACT (a brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual).

Although there already exist tools to assist in understanding the behavior of software systems when no complete and consistent design models are available, these tools generate a large volume of information. One approach to deal with this problem is information hiding. This technical memorandum presents a prototype which implements this technique to reverse engineer dynamic models from Java software systems. These models are represented using Unified Modeling Language (UML) sequence diagrams. Such diagrams show the interactions, in terms of messages or information transfers, between the operational nodes of a system, arranged in a time sequence. Information hiding is achieved by reconstructing the sequence diagrams at various levels of abstraction. The interactions between the operational nodes of a system can be displayed at a low level, i.e., object level. However, related operational nodes can also be regrouped into higher level structures, i.e., packages. The proposed approach was implemented in Eclipse, an extensible integrated development environment (IDE). The objective is to complement the behavioral views reverse engineered by the implemented prototype with structure views generated by other tools.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus, e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus-identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

Software architecture recovery, program comprehension, program understanding tools, reverse engineering, prototype.